February 1981

Getting Started With the Numeric Data Processor

Bill Rash
Microcomputer Applications

INTRODUCTION

This is an application note on using numerics in Intel's iAPX 86 or iAPX 88 microprocessor family. The numerics implemented in the family provide instruction level support for high-precision integer and floating point data types with arithmetic operations like add, subtract, multiply, divide, square root, power, log and trigonometrics. These features are provided by members of the iAPX 86 or iAPX 88 family called numeric data processors.

Rather than concentrate on a narrow, specific application, the topics covered in this application note were chosen for generality across many applications. The goal is to provide sufficient background information so that software and hardware engineers can quickly move beyond needs specific to the numeric data processor and concentrate on the special needs of their application. The material is structured to allow quick identification of relevant material without reading all the material leading up to that point. Everyone should read the introduction to establish terminology and a basic background.

IAPX 86,88 BASE

The numeric data processor is based on an 8088 or 8086 microprocessor. The 8086 and 8088 are general purpose microprocessors, designed for general data processing applications. General applications need fast, efficient data movement and program control instructions. Actual arithmetic on data values is simple in general applications. The 8086 and 8088 fulfill these needs in a low cost, effective manner.

However, some applications need more powerful arithmetic instructions and data types than a general purpose data processor provides. The real world deals in fractional values and requires arithmetic operations like square root, sine, and logarithms. Integer data types and their operations like add, subtract, multiply, and divide may not meet the needs for accuracy, speed, and ease of use.

Such functions are not simple or inexpensive. The general data processor does not provide these features due to their cost to other less-complex applications that do not need such features. A special processor is required, one which is easy to use and has a high level of support in hardware and software.

The numeric data processor provides these features. It supports the data types and operations needed and allows use of all the current hardware and software support for the iAPX 86/10 and 88/10 microprocessors.

The iAPX 86 and iAPX 88 provide two implementations of a numeric data processor. Each offers different tradeoffs in performance, memory size, and cost.

One alternative uses a special hardware component, the 8087 numeric processor extension, while the other is based on software, the 8087 emulator. Both component and software emulator add the extra numerics data types and operations to the 8086 or 8088.

The component and its software emulator are completely compatible.

Nomenclature

Table one shows several possible configurations of the iAPX 86 and iAPX 88 microprocessor family. The choice of configuration will be decided by the needs of the application for cost and performance in the areas of general data processing, numerics, and I/O processing. The combination of an 8086 or 8088 with an 8087 is called an iAPX 86/20 or 88/20 numeric data processor. For applications requiring high I/O bandwidths and numeric performance, a combination of 8086, 8087 and 8089 is an iAPX 86/21 numerics and I/O data processor. The same system with an 8088 CPU for smaller size and lower cost, due to the smaller 8-bit wide system data bus, is referred to as an iAPX 88/21. Each 8089 in the system is designated in the units digit of the system designation. The term 86/2X or 88/2X refers to a numeric data processor with any number of 8089s.

Throughout this application note, I will use the terms NDP, numeric data processor, 86/2X, and 88/2X synonymously. Numeric processor extension and NPX are also synonymous for the functions of either the 8087 component or 8087 emulator. The term numeric instruction or numeric data type refers to an instruction or data type made available by the NPX. The term host will refer to either the 8086 or 8088 microprocessor.

Table 1. Components Used in I/APX 86,88 Configurations

System Name	8086	8087	8088	8089
iAPX 86/10	1			
iAPX 86/11	1			1
iAPX 86/12	1			2
iAPX 86/20	1	1		
iAPX 86/21	1	1		1
iAPX 86/22	1	1		2
iAPX 88/10			1	
iAPX 88/11			1	1
iAPX 88/12			1	2
iAPX 88/20		1	1	
iAPX 88/21		1	1	1
iAPX 88/22		1	1	2

NPX OVERVIEW

The 8087 is a coprocessor extension available to iAPX 86/1X or iAPX 88/1X maximum mode microprocessor systems. (See page 7). The 8087 adds hardware support for floating point and extended precision integer data types, registers, and instructions. Figure 1 shows the register set available to the NDP. On the next page, the seven data types available to numeric instructions are listed (Fig 2). Each data type has a load and store instruction. Independent of whether an 8087 or its emulator are used, the registers and data types all appear the same to the programmer.

All the numeric instructions and data types of the NPX are used by the programmer in the same manner as the general data types and instructions of the host.

The numeric data formats and arithmetic operations provided by the 8087 conform to the proposed IEEE Microprocessor Floating Point Standard. All the proposed IEEE floating point standard algorithms, exception detection, exception handling, infinity arithmetic and rounding controls are implemented.[1]

The numeric registers of the NPX are provided for fast, easy reference to values needed in numeric calculations. All numeric values kept in the NPX register file are held in the 80-bit temporary real floating point format which is the same as the 80-bit temporary real data type.

All data types are converted to the 80-bit register file format when used by the NPX. Load and store instructions automatically convert between the memory operand data type and the register file format for all numeric data types. The numeric load instruction specifies the format in which the memory operand is expected and which addressing mode to use.

All host base registers, index registers, segment registers, and addressing modes are available for locating numeric operands. In the same manner, the store instruction also specifies which data type to use and where the value is located when stored into memory.

Selecting Numeric Data Types

As figure 2 shows, the numeric data types are of different lengths and domains (real or integer). Each numeric data type is provided for a specific function, they are:

16-bit word integers	—Index values, loop counts, and small program control values
32-bit short integers	—Large integer general computation
64-bit long integers	—Extended range integer computation
18-digit packed decimal	—Commercial and decimal conversion arithmetic
32-bit short real	—Reduced range and accuracy is traded for reduced memory requirements
64-bit long real	—Recommended floating point variable type
80-bit temporary real	—Format for intermediate or high precision calculations

Referencing memory data types in the NDP is not restricted to load and store instructions. Some arithmetic operations can specify a memory operand in one of four possible data types. The numeric instructions compare, add, subtract, subtract reversed, multiply, divide, and divide reversed can specify a memory operand to be either a 16-bit integer, 32-bit integer, 32-bit real, or 64-bit real value. As with the load and store operations, the arithmetic instruction specifies the address and expected format of the memory operand.

The remaining arithmetic operations: square root, modulus, tangent, arctangent, logarithm, exponentiate, scale power, and extract power use only register operands.

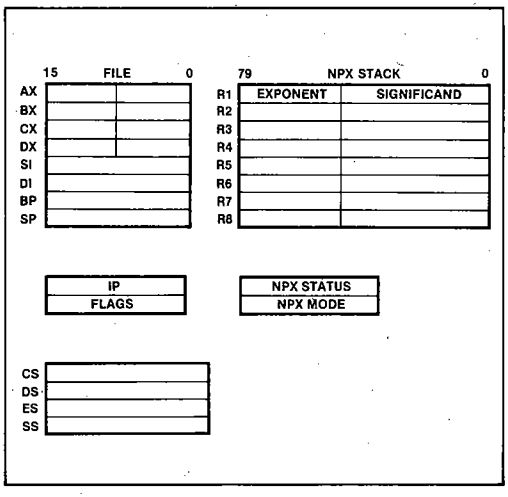


Figure 1. NDP Register Set for iAPX 86/20, 88/20

[1]"An Implementation Guide to a Proposed Standard for Floating Point" by Jerome Coonen in *Computer*, Jan. 1980 or the Oct. 1979 issue of *ACM SIGNUM*, for more information on the standard.

The register set of the host and 8087 are in separate components. Direct transfer of values between the two register sets in one instruction is not possible. To transfer values between the host and numeric register sets, the value must first pass through memory. The memory format of a 16-bit short integer used by the NPX is identical to that of the host, ensuring fast, easy transfers.

Since an 8086 or 8088 does not provide single instruction support for the remaining numeric data types, host programs reading or writing these data types must conform to the bit and byte ordering established by the NPX.

Writing programs using numeric instructions is as simple as with the host's instructions. The numeric instructions are simply placed in line with the host's instructions. They are executed in the same order as they appear in the instruction stream. Numeric instructions follow the same form as the host instructions. Figure 2 shows the ASM 86/88 representations for different numeric instructions and their similarity to host instructions.

```
FILD        VALUE
FIADD       TABLE [BX]
FADD        ST,ST(1)
```

8087 EMULATOR OVERVIEW

The NDP has two basic implementations, an 8087 component or with its software emulator (E8087). The decision to use the emulator or component has no effect on programs at the source level. At the source level, all instructions, data types, and features are used the same way.

The emulator requires all numeric instruction opcodes to be replaced with an interrupt instruction. This replacement is performed by the LINK86 program. Interrupt vectors in the host's interrupt vector table will point to numeric instruction emulation routines in the 8087 software emulator.

When using the 8087 emulator, the linker changes all the 2-byte wait-escape, nop-escape, wait-segment override, or nop-segment override sequences generated by an assembler or compiler for the 8087 component with a 2-byte interrupt instruction. Any remaining bytes of the numeric instruction are left unchanged.

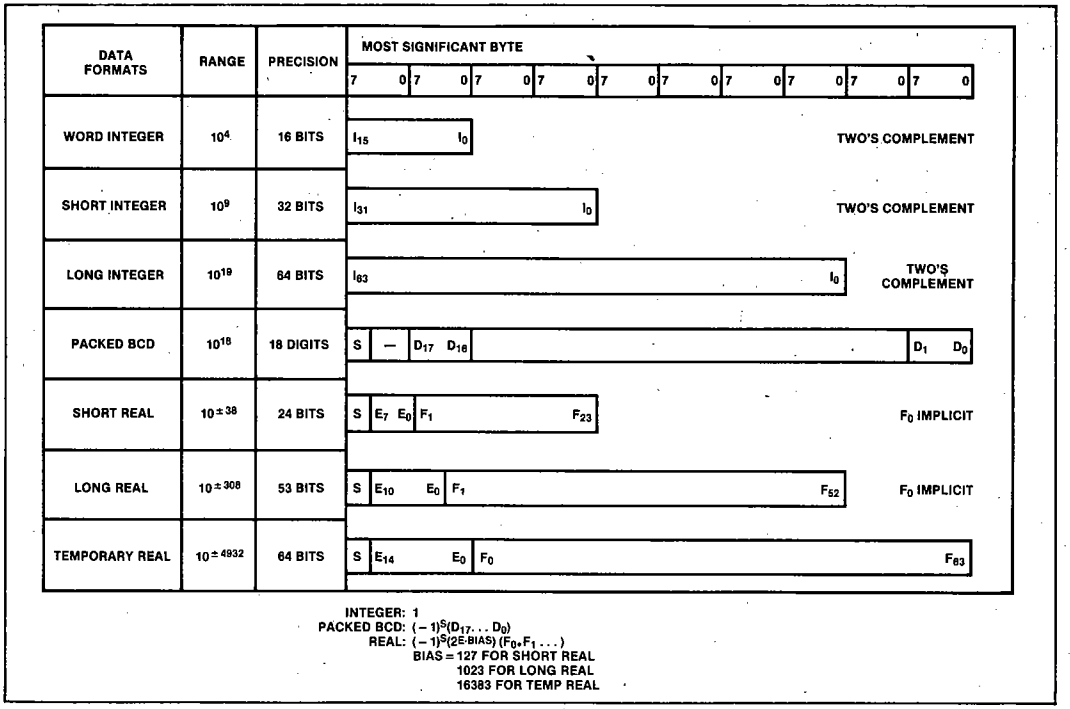


Figure 2. NPX Data Types

When the host encounters numeric and emulated instruction, it will execute the software interrupt instruction formed by the linker. The interrupt vector table will direct the host to the proper entry point in the 8087 emulator. Using the interrupt return address and CPU register set, the host will decode any remaining part of the numeric instruction, perform the indicated operation, then return to the next instruction following the emulated numeric instruction.

One copy of the 8087 emulator can be shared by all programs in the host.

The decision to use the 8087 or software emulator is made at link time, when all software modules are brought together. Depending on whether an 8087 or its software emulator is used, a different group of library modules are included for linking with the program.

If the 8087 component is used, the libraries do not add any code to the program, they just satisfy external references made by the assembler or compiler. Using the emulator will not increase the size of individual modules; however, other modules requiring about 16K bytes that implement the emulator will be automatically added.

Selecting between the emulator or the 8087 can be very easy. Different versions of submit files performing the link operation can be used to specify the different set of library modules needed. Figure 3 shows an example of two different submit files for the same program using the NPX with an 8087 or the 8087 emulator.

```
8087 BASED LINK/LOCATE COMMANDS

LINK86 :F1:PROG.OBJ, IO.LIB, 8087.LIB TO
       :F1:PROG.LNK
LOC86  :F1:PROG.LNK TO :F1:PROG


SOFTWARE EMULATOR BASED
LINK/LOCATE COMMANDS

LINK86 :F1:PROG.OBJ,  IO.LIB, E8087.LIB,
       E8087 TO :F1:PROG.LNK
LOC86  :F1:PROG.LNK TO :F1:PROG
```

Figure 3. Submit File Example

iSBC 337™ MULTIMODULE™ Overview

The benefits of the NPX are not limited to systems which left board space for the 8087 component or memory space for its software emulator. Any maximum mode iAPX 86/1X or iAPX 88/1X system can be upgraded to a numeric processor. The iSBC 337 MULTIMODULE is designed for just this function. The iSBC 337 provides a socket for the host microprocessor and an 8087. A 40-pin plug is provided on the underside of the 337 to plug into the original host's socket, as shown in Figure 4. Two other pins on the underside of the MULTIMODULE allow easy connection to the 8087 INT and RQ/GT1 pins.

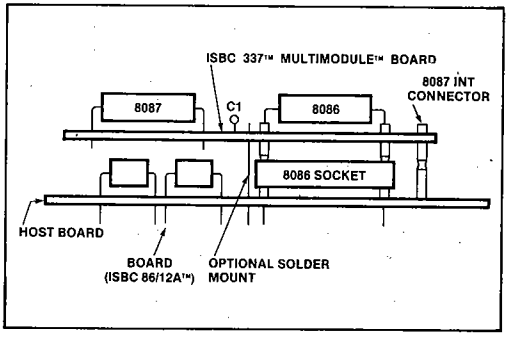


Figure 4. MULTIMODULE™ Math Mounting Scheme

CONSTRUCTING AN iAPX 86/2X OR iAPX 88/2X SYSTEM

This section will describe how to design a microprocessor system with the 8087 component. The discussion will center around hardware issues. However, some of the hardware decisions must be made based upon how the software will use the NPX. To better understand how the 8087 operates as a local bus master, we shall cover how the coprocessor interface works later in this section.

Wiring up the 8087

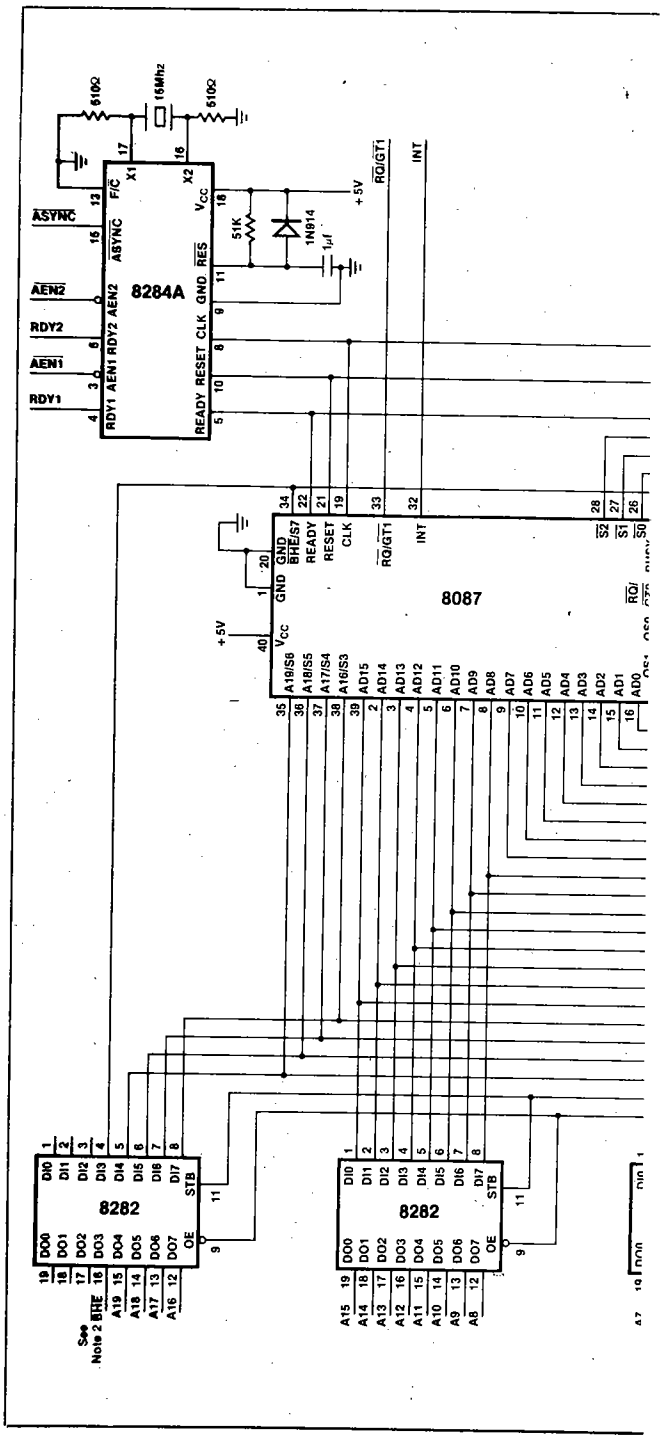
The 8087 can be designed into any 86/1X or 88/1X system operating in maximum mode. Such a system would be designated an 86/2X or 88/2X. Figure 5 shows the local bus interconnections for an iAPX 86/20 (or iAPX 88/20) system. The 8087 shares the maximum mode host's multiplexed address/data bus, status signals, queue status signals, ready status signal, clock and reset signal. Two dedicated signals, BUSY and INT, inform the host of current 8087 status. The 10K pull-down resistor on the BUSY signal ensures the host will always see a "not busy" status if an 8087 is not installed.

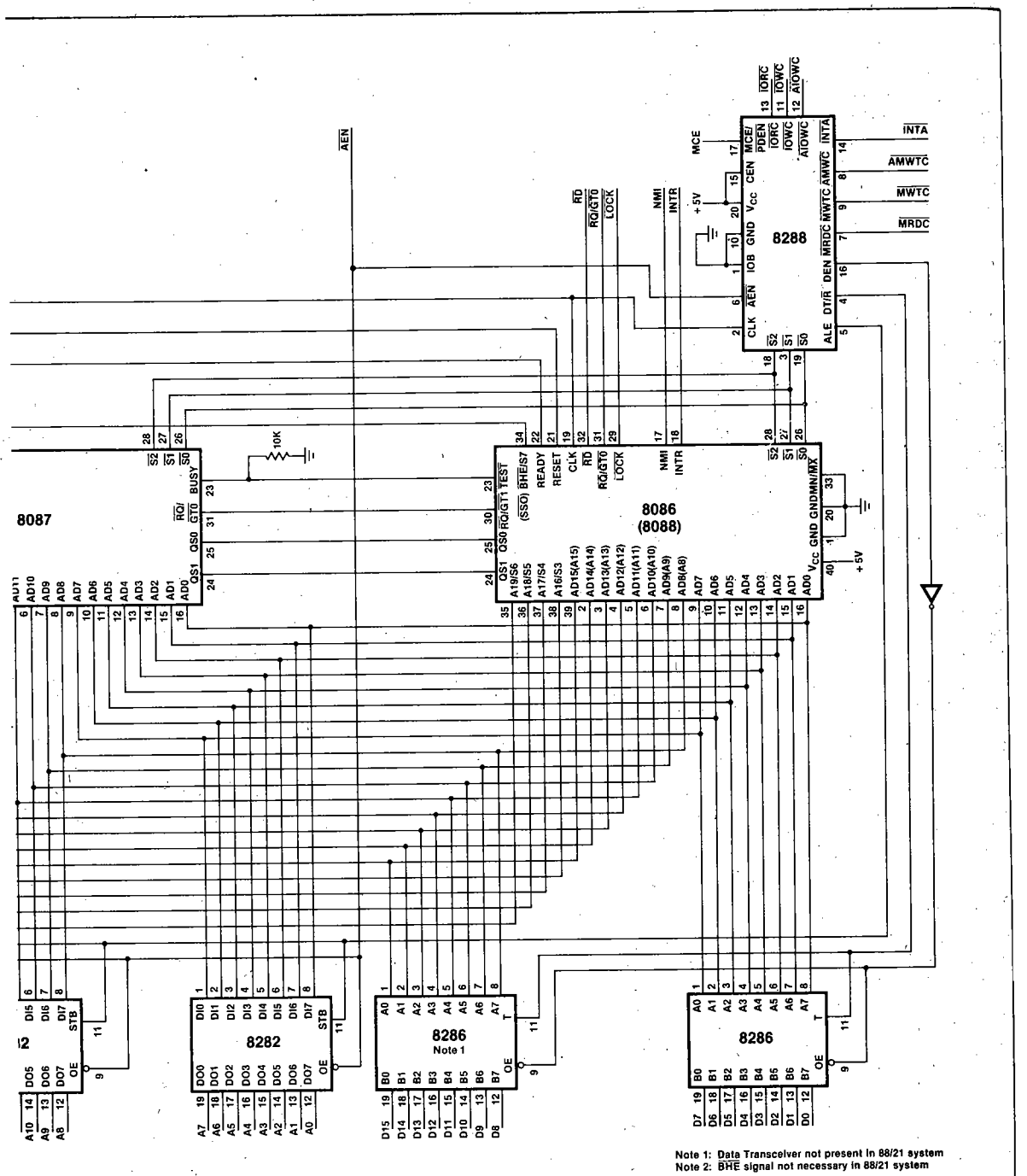
Adding the 8087 to your design has a minor effect on hardware timing. The 8087 has the exact same timing and equivalent DC and AC drive characteristics as a host or IOP on the local bus. All the local bus logic, such as clock, ready, and interface logic is shared.

The 8087 adds 15 pF to the total capacitive loading on the shared address/data and status signals. Like the 8086 or 8088, the 8087 can drive a total of 100 pF capacitive load above its own self load and sink 2.0 mA DC current on these pins. This AC and DC drive is sufficient for an 86/21 system with two sets of data transceivers, address latches, and bus controllers for two separate busses, an on-board bus and an off-board MULTIBUS™ using the 8289 bus arbiter.

Later in this section, what to do with the 8087 INT and RQ/GT pins, is covered.

It is possible to leave a prewired 40-pin socket on the board for the 8087. Adding the 8087 to such a system is as easy as just plugging it in. If a program attempts to execute any numeric instructions without the 8087 installed, they will be simply treated as NOP instructions by the host. Software can test for the existence of the 8087 by initializing it and then storing the control word. The program of Figure 6 illustrates this technique.




8288
8086
(8088)
8087
8282
8286
Note 1
8286
AEN
MCE
INTA
AMWTC
MWTC
MRDC
10K
+5V
+5V
Note 1: Data Transceiver not present in 88/21 system
Note 2: BHE signal not necessary in 88/21 system

207865-001

WHAT IS THE iAPX 86, 88 COPROCESSOR INTERFACE?

The idea of a coprocessor is based on the observation that hardware specially designed for a function is the fastest, smallest, and cheapest implementation. But, it is too expensive to incorporate all desired functions in general purpose hardware. Few applications could use all the functions. To build fast, small, economical systems, we need some way to mix and match components supporting specialized functions.

Purpose of the Coprocessor Interface

The coprocessor interface of the general purpose 8086 or 8088 microprocessor provides a way to attach specialized hardware in a simple, elegant, and efficient manner. Because the coprocessor hardware is specialized, it can perform its job much faster than any general purpose CPU of similar size and cost. The coprocessor interface simply requires connection to the host's local address/data, status, clock, ready, reset, test and request/grant signals. Being attached to the host's local bus gives the coprocessor access to all memory and I/O resources available to the host.

The coprocessor is independent of system configuration. Using the local bus as the connection point to the host isolates the coprocessor from the particular system configuration, since the timing and function of local bus signals are fixed.

Software's View of the Coprocessor

The coprocessor interface allows specialized hardware to appear as an integral part of the host's architecture controlled by the host with special instructions. When the host encounters these special instructions, both the host and coprocessor recognize them and work together to perform the desired function. No status polling loops or command stuffing sequences are required by software to operate the coprocessor.

More information is available to a coprocessor than simply an instruction opcode and a signal to begin execution. The host's coprocessor interface can read a value from memory, or identify a region of memory the coprocessor should use while performing its function. All the addressing modes of the host are available to identify memory based operands to the coprocessor.

Concurrent Execution of Host and Coprocessor

After the coprocessor has started its operation, the host may continue on with the program, executing it in parallel while the coprocessor performs the function started earlier. The parallel operation of the coprocessor does not normally affect that of the host unless the coprocessor must reference memory or I/O-based operands. When the host releases the local bus to the coprocessor, the host may continue to execute from its internal instruction queue. However, the host must stop when it also needs the local bus currently in use by the coprocessor. Except for the stolen memory cycle, the operation of the coprocessor is transparent to the host.

This parallel operation of host and coprocessor is called concurrent execution. Concurrent execution of instructions requires less total time then a strictly sequential execution would. System performance will be higher with concurrent execution of instructions between the host and coprocessor.

SYNCHRONIZATION

In exchange for the higher system performance made available by concurrent execution, programs must provide what is called synchronization between the host and coprocessor. Synchronization is necessary whenever the host and coprocessor must use information available from the other. Synchronization involves either the host or coprocessor waiting for the other to finish an operation currently in progress. Since the host executes the program, and has program control instructions like jumps, it is given responsibility for synchronization. To meet this need, a special host instruction exists to synchronize host operation with a coprocessor.

```
;
; Test for the existence of an 8087 in the system. This code will always recognize an 8087
; independent of the TEST pin usage on the host. No deadlock is possible. Using the 8087
; emulator will not change the function of this code since ESC instructions are used. The word
; variable control is used for communication between the 8087 and the host. Note: if an 8087 is
; present, it will be initialized. Register ax is not transparent across this code.
;
ESC   28, bx          ; FNINIT if 8087 is present . The contents of bx is irrelevant
XOR   ax, ax          ; These two instructions insert delay while the 8087 initializes itself
MOV   control, ax     ; Clear intial control word value
ESC   15, control     ; FNSTCW if 8087 is present
OR    ax, control     ; Control = 03ffh if 8087 present
JZ    no_8087         ; Jump if no 8087 is present
```

Figure 6. Test for Existence of an 8087

The host coprocessor synchronization instruction, called "WAIT", uses the TEST pin of the host. The coprocessor can signal that it is still busy to the host via this pin. Whenever the host executes a wait instruction, it will stop program execution while the TEST input is active. When the TEST pin becomes inactive, the host will resume program execution with the next instruction following the WAIT. While waiting on the TEST pin, the host can be interrupted at 5 clock intervals; however, after the TEST pin becomes inactive, the host will immediately execute the next instruction, ignoring any pending interrupts between the WAIT and following instruction.

COPROCESSOR CONTROL

The host has the responsibility for overall program control. Coprocessor operation is initiated by special instructions encountered by the host. These instructions are called "ESCAPE" instructions. When the host encounters an ESCAPE instruction, the coprocessor is expected to perform the action indicated by the instruction. There are 576 different ESCAPE instructions, allowing the coprocessor to perform many different actions.

The host's coprocessor interface requires the coprocessor to recognize when the host has encountered an ESCAPE instruction. Whenever the host begins executing a new instruction, the coprocessor must look to see if it is an ESCAPE instruction. Since only the host fetches instructions and executes them, the coprocessor must monitor the instructions being executed by the host.

Host Queue Tracking

The host can fetch an instruction at a variable length time before the host executes the instruction. This is a characteristic of the instruction queue of an 8086 or 8088 microprocessor. An instruction queue allows prefetching instructions during times when the local bus would be otherwise idle. The end benefit is faster execution time of host instructions for a given memory bandwidth.

The host does not externally indicate which instruction it is currently executing. Instead, the host indicates when it fetches an instruction and when, some time later, an opcode byte is decoded and executed. To identify the actual instruction the host fetched from its queue, the coprocessor must also maintain an instruction stream identical to the host's.

Instructions can be fetched in byte or word increments, depending on the type of host and the destination address of jump instructions executed by the host. When the host has filled its queue, it stops prefetching instructions. Instructions are removed from the queue a byte at a time for decoding and execution. When a jump occurs, the queue is emptied. The coprocessor follows these actions in the host by monitoring the host's bus status, queue status, and data bus signals. Figure 7 shows how the bus status signals and queue status signals are encoded.

IGNORING I/O PROCESSORS

The host is not the only local bus master capable of fetching instructions. An Intel 8089 IOP can generate instruction fetches on the local bus in the course of executing a channel program in system memory. In this case, the status signals S2, S1, and S0 generated by the IOP are identical to those of the host. The coprocessor must not interpret these instruction prefetches as going to the host's instruction queue. This problem is solved with a status signal called S6. The S6 signal identifies when the local bus is being used by the host. When the host is the local bus master, S6 = 0 during T2 and T3 of the memory cycle. All other bus masters must set S6 = 1 during T2 and T3 of their instruction prefetch cycles. Any coprocessor must ignore activity on the local bus when S6 = 1.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Function	QS1	QS0	Host Function	Coprocessor Activity
0	0	0	Interrupt Acknowledge	0	0	No Operation	No Queue Activity
0	0	1	Read I/O Port	0	1	First Byte	Decode Opcode Byte
0	1	0	Write I/O Port	1	0	Empty Queue	Empty Queue
0	1	1	Halt	1	1	Subsequent Byte	Flush Byte or if 2nd Byte of Escape Decode it
1	0	0	Code Fetch				
1	0	1	Read Data Memory				
1	1	0	Write Data Memory				
1	1	1	Idle				

Figure 7.

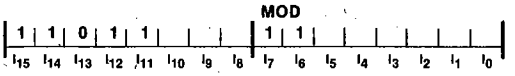
DECODING ESCAPE INSTRUCTIONS

To recognize ESCAPE instructions, the coprocessor must examine all instructions executed by the host. When the host fetches an instruction byte from its internal queue, the coprocessor must do likewise.

The queue status state, fetch opcode byte, identifies when an opcode byte is being examined by the host. At the same time, the coprocessor will check if the byte fetched from its internal instruction queue is an ESCAPE opcode. If the instruction is not an ESCAPE, the coprocessor will ignore it. The queue status signals for fetch subsequent byte and flush queue let the coprocessor track the host's queue without knowledge of the length and function of host instructions and addressing modes.

Escape Instruction Encoding

All ESCAPE instructions start with the high-order 5-bits of the instruction being 11011. They have two basic forms. The non-memory form, listed here, initiates some activity in the coprocessor using the nine available bits of the ESCAPE instruction to indicate which function to perform.



Memory reference forms of the ESCAPE instruction, shown in Figure 8, allow the host to point out a memory operand to the coprocessor using any host memory addressing mode. Six bits are available in the memory reference form to identify what to do with the memory operand. Of course, the coprocessor may not recognize all possible ESCAPE instructions, in which case it will simply ignore them.

Memory reference forms of ESCAPE instructions are identified by bits 7 and 6 of the byte following the ESCAPE opcode. These two bits are the MOD field of the 8086 or 8088 effective address calculation byte. They, together with the R/M field, bits 2 through 0, determine the addressing mode and how many subsequent bytes remain in the instruction.

Host's Response to an Escape Instruction

The host performs one of two possible actions when encountering an ESCAPE instruction: do nothing or calculate an effective address and read a word value beginning at that address. The host ignores the value of the word read. ESCAPE instructions change no registers in the host other than advancing IP. So, if there is no coprocessor, or the coprocessor ignores the ESCAPE instruction, the ESCAPE instruction is effectively a NOP to the host. Other than calculating a memory address and reading a word of memory, the host makes no other assumptions regarding coprocessor activity.

The memory reference ESCAPE instructions have two purposes: identify a memory operand and for certain instructions, transfer a word from memory to the coprocessor.

COPROCESSOR INTERFACE TO MEMORY

The design of a coprocessor is considerably simplified if it only requires reading memory values of 16 bits or less. The host can perform all the reads with the coprocessor latching the value as it appears on the data bus at the end of T3 during the memory read cycle. The coprocessor need never become a local bus master to read or write additional information.

If the coprocessor must write information to memory, or deal with data values longer than one word, then it must save the memory address and be able to become a local bus master. The read operation performed by the host in the course of executing the ESCAPE instruction places the 20-bit physical address of the operand on the address/data pins during T1 of the memory cycle. At this time the coprocessor can latch the address. If the coprocessor instruction also requires reading a value, it will appear on the data bus during T3 of the memory read. All other memory bytes are addressed relative to this starting physical address.

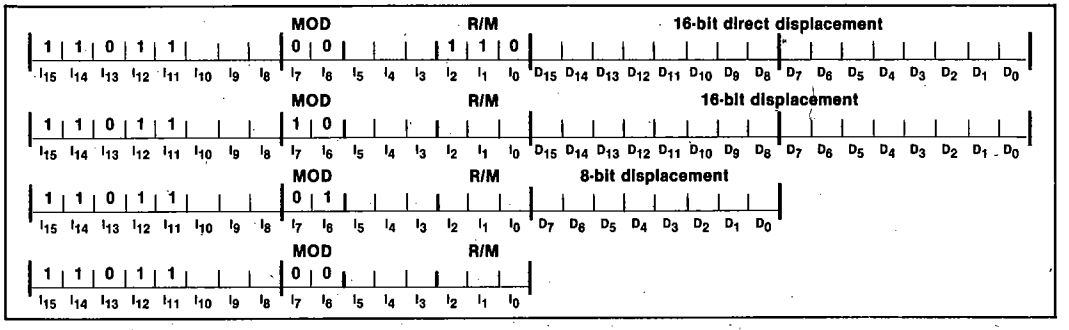


Figure 8. Memory Reference Escape Instruction Forms

Whether the coprocessor becomes a bus master or not, if the coprocessor has memory reference instruction forms, it must be able to identify the memory read performed by the host in the course of executing an ESCAPE instruction.

Identifying the memory read is straightforward, requiring all the following conditions to be met:

1) A MOD value of 00, 01, or 10 in the second byte of the ESCAPE instruction executed by the host.
2) This is the first data read memory cycle performed by the host after it encountered the ESCAPE instruction. In particular, the bus status signals S2–S0 will be 101 and S6 will be 0.

The coprocessor must continue to track the instruction queue of the host while it calculates the memory address and reads the memory value. This is simply a matter of following the fetch subsequent byte status commands occurring on the queue status pins.

HOST PROCESSOR DIFFERENCES

A coprocessor must be aware of the bus characteristics of the host processor. This determines how the host will read the word operand of a memory reference ESCAPE instruction. If the host is an 8088, it will always perform two byte reads at sequential addresses. But if the host is an 8086, it can either perform a single word read or two byte reads to sequential addresses.

The 8086 places no restrictions on the alignment of word operands in memory. It will automatically perform two byte operations for word operands starting at an odd address. The two operations are necessary since the two bytes of the operand exist in two different memory words. The coprocessor should be able to accept the two possible methods of reading a word value on the 8086.

A coprocessor can determine whether the 8086 will perform one or two memory cycles as part of the current ESCAPE instruction execution. The AD0 pin during T1 of the first memory read by the host tells if this is the only read to be performed as part of the ESCAPE instruction. If this pin is a 1 during T1 of the memory cycle, the 8086 will immediately follow this memory read cycle with another one at the next byte address.

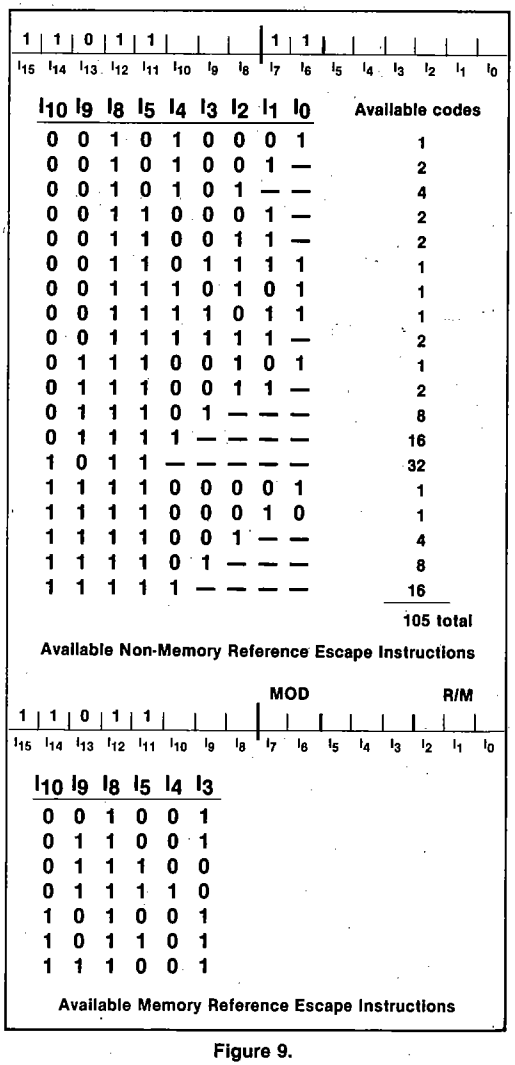
Coprocessor Interface Summary

The host ESCAPE instructions, coprocessor interface, and WAIT instruction allow easy extension of the host's architecture with specialized processors. The 8087 is such a processor, extending the host's architecture as seen by the programmer. The specialized hardware provided by the 8087 can greatly improve system performance economically in terms of both hardware and software for numerics applications.

The next section examines how the 8087 uses the coprocessor interface of the 8086 or 8088.

8087 COPROCESSOR OPERATION

The 8086 or 8088 ESCAPE instructions provide 64 memory reference opcodes and 512 non-memory reference opcodes. The 8087 uses 57 of the memory reference forms and 406 of the non-memory reference forms. Figure 9 shows the ESCAPE instructions not used by the 8087.



I10	I9	I8	I5	I4	I3	I2	I1	I0	Available codes
0	0	1	0	1	0	0	0	1	1
0	0	1	0	1	0	0	1	—	2
0	0	1	0	1	0	1	—	—	4
0	0	1	1	0	0	0	1	—	2
0	0	1	1	0	0	1	1	—	2
0	0	1	1	0	1	1	1	1	1
0	0	1	1	1	0	1	0	1	1
0	0	1	1	1	1	0	1	1	1
0	0	1	1	1	1	1	1	—	2
0	1	1	1	0	0	1	0	1	1
0	1	1	1	0	0	1	1	—	2
0	1	1	1	0	1	—	—	—	8
0	1	1	1	1	—	—	—	—	16
1	0	1	1	—	—	—	—	—	32
1	1	1	1	0	0	0	0	1	1
1	1	1	1	0	0	0	1	0	1
1	1	1	1	0	0	1	—	—	4
1	1	1	1	0	1	—	—	—	8
1	1	1	1	1	—	—	—	—	16
									105 total

Available Non-Memory Reference Escape Instructions

I10	I9	I8	I5	I4	I3
0	0	1	0	0	1
0	1	1	0	0	1
0	1	1	1	0	0
0	1	1	1	1	0
1	0	1	0	0	1
1	0	1	1	0	1
1	1	1	0	0	1

Available Memory Reference Escape Instructions

Figure 9.

Using the 8087 With Custom Coprocessors

Custom coprocessors, a designer may care to develop, should limit their use of ESCAPE instructions to those not used by the 8087 to prevent ambiguity about whether any one ESCAPE instruction is intended for a numerics or other custom coprocessor. Using any escape instruction for a custom coprocessor may conflict with opcodes chosen for future Intel coprocessors.

Operation of an 8087 together with other custom coprocessors is possible under the following constraints:

1) All 8087 errors are masked. The 8087 will update its opcode and instruction address registers for the unused opcodes. Unused memory reference instructions will also update the operand address value. Such changes in the 8087 make software-defined error handling impossible.
2) If the coprocessors provide a BUSY signal, they must be ORed together for connection to the host TEST pin. When the host executes a WAIT instruction, it does not know which coprocessor will be affected by the following ESCAPE instruction. In general, all coprocessors must be idle before executing the ESCAPE instruction.

Operand Addressing by the 8087

The 8087 has seven different memory operand formats. Six of them are longer than one word. All are an even number of bytes in length and are addressed by the host at the lowest address word.

When the host executes a memory reference ESCAPE instruction intended to cause a read operation by the 8087, the host always reads the low-order word of any 8087 memory operand. The 8087 will save the address and data read. To read any subsequent words of the operand, the 8087 must become a local bus master.

When the 8087 has the local bus, it increments the 20-bit physical address it saved to address the remaining words of the operand.

When the ESCAPE instruction is intended to cause a write operation by the 8087, the 8087 will save the address but ignore the data read. Eventually, it will get control of the local bus, then perform successive write, increment address operations writing the entire data value.

8087 OPERATION IN IAPX 86,88 SYSTEMS

The 8087 will work with either an 8086 or 8088 host. The identity of the host determines the width of the local bus path. The 8087 will identify the host and adjust its use of the data bus accordingly; 8 bits for an 8088 or 16 bits for an 8086. No strapping options are required by the 8087; host identification is automatic.

The 8087 identifies the host each time the host and 8087 are reset via the RESET pin. After the reset signal goes inactive, the host will begin instruction execution at memory address $FFFF0_{16}$.

If the host is an 8086 it will perform a word read at that address; an 8088 will perform a byte read.

The 8087 monitors pin 34 on the first memory cycle after power up. If an 8086 host is used, pin 34 will be the BHE signal, which will be low for that memory cycle. For an 8088 host, pin 34 will be the SS0 signal, which will be high during T1 of the first memory cycle. Based on this signal, the 8087 will then configure its data bus width to match that of the host local bus.

For 88/2X systems, pin 34 of the 8087 may be tied to V_{CC} if not connected to the 8088 SS0 pin.

The width of the data bus and alignment of data operands has no effect, except for execution time and number of memory cycles performed, on 8087 instructions. A numeric program will always produce the same results on an 86/2X or 88/2X with any operand alignment. All numeric operands have the same relative byte orderings independent of the host and starting address.

The byte alignment of memory operands can affect the performance of programs executing on an 86/2X. If a word operand, or any numeric operand, starts on an odd-byte address, more memory cycles are required to access the operand than if the operand started on an even address. The extra memory cycles will lower system performance.

The 86/2X will attempt to minimize the number of extra memory cycles required for odd-aligned operands. In these cases, the 8087 will perform first a byte operation, then a series of word operations, and finally a byte operation.

88/2X instruction timings are independent of operand alignment, since byte operations are always performed. However, it is recommended to align numeric operands on even boundaries for maximum performance in case the program is transported to an 86/2X.

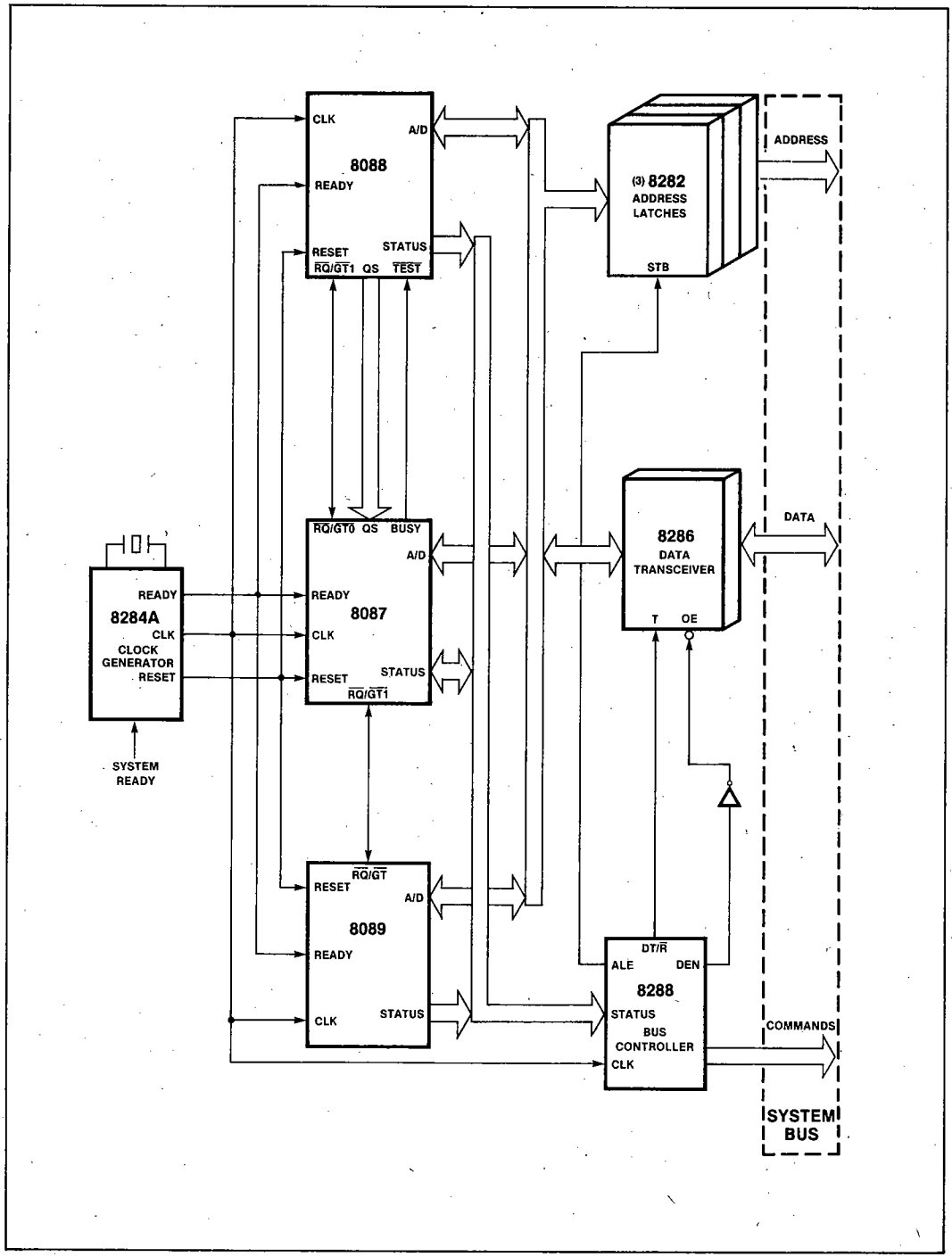


Figure 10. iAPX 88/21

RQ/GT CONNECTION

Two decisions must be made when connecting the 8087 to a system. The first is how to interconnect the RQ/GT signals of all local bus masters. The RQ/GT decision affects the response time to service local bus requests from other local bus masters, such as an 8089 IOP or other coprocessor. The interrupt connection affects the response time to service an interrupt request and how user-interrupt handlers are written. The implications of how these pins are connected concern both the hardware designer and programmer and must be understood by both.

The RQ/GT issue can be broken into three general categories, depending on system configuration: 86/20 or 88/20, 86/21 or 88/21, and 86/22 or 88/22. Remote operation of an IOP is not effected by the 8087 RQ/GT connection.

iAPX 86/20, 88/20

For an 86/20 or 88/20 just connect the RQ/GT0 pin of the 8087 to RQ/GT1 of the host (see Figure 5), and skip forward to the interrupt discussion on page 15.

iAPX 86/21, 88/21

For an 86/21 or 88/21, connect RQ/GT0 of the 8087 to RQ/GT1 of the host, connect RQ/GT of the 8089 to RQ/GT1 of the 8087 (see Figure 10, page 12), and skip forward to the interrupt discussion on page 15.

The RQ/GT1 pin of the 8087 exists to provide one I/O processor with a low maximum wait time for the local bus. The maximum wait times to gain control of the local bus for a device attached to RQ/GT1 of an 8087 for an 8086 or 8088 host are shown in Table 2. These numbers are all dependent on when the host will release the local bus to the 8087.

As Table 2 implies, three factors determine when the host will release the local bus:

1) What type of host is there, an 8086 or 8088?

2) What is the current instruction being executed?

3) How is the lock prefix being used?

An 8086 host will not release the local bus between the two consecutive byte operations performed for odd-aligned word operands. The 8088, in contrast, will never release the local bus between the two bytes of a word transfer, independent of its byte alignment.

Host operations such as acknowledging an interrupt will not release the local bus for several bus cycles.

Using a lock prefix in front of a host instruction prevents the host from releasing the local bus during the execution of that instruction.

8087 RQ/GT Function

The presence of the 8087 in the RQ/GT path from the IOP to the host has little effect on the maximum wait time seen by the IOP when requesting the local bus. The 8087 adds two clocks of delay to the basic time required by the host. This low delay is achieved due to a preemptive protocol implemented by the 8087 on RQ/GT1.

The 8087 always gives higher priority to a request for the local bus from a device attached to its RQ/GT1 pin than to a request generated internally by the 8087. If the 8087 currently owns the local bus and a request is made to its RQ/GT1 pin, the 8087 will finish the current memory cycle and release the local bus to the requestor. If the request from the devices arrives when the 8087 does not own the local bus, then the 8087 will pass the request on to the host via its RQ/GT0 pin.

Table 2. Worst Case Local Bus Request Wait Times in Clocks

System Configuration	No Locked Instructions	Only Locked Exchange	Other Locked Instructions
iAPX 86/21 even aligned words	15_1	35_1	max (15_1, *)
iAPX 86/21 odd aligned words	15_1	43_2	max (43_2, *)
iAPX 88/21	15_1	43_2	max (43_2, *)

Notes: 1. Add two clocks for each wait state inserted per bus cycle
2. Add four clocks for each wait state inserted per bus cycle
* Execution time of longest locked instruction

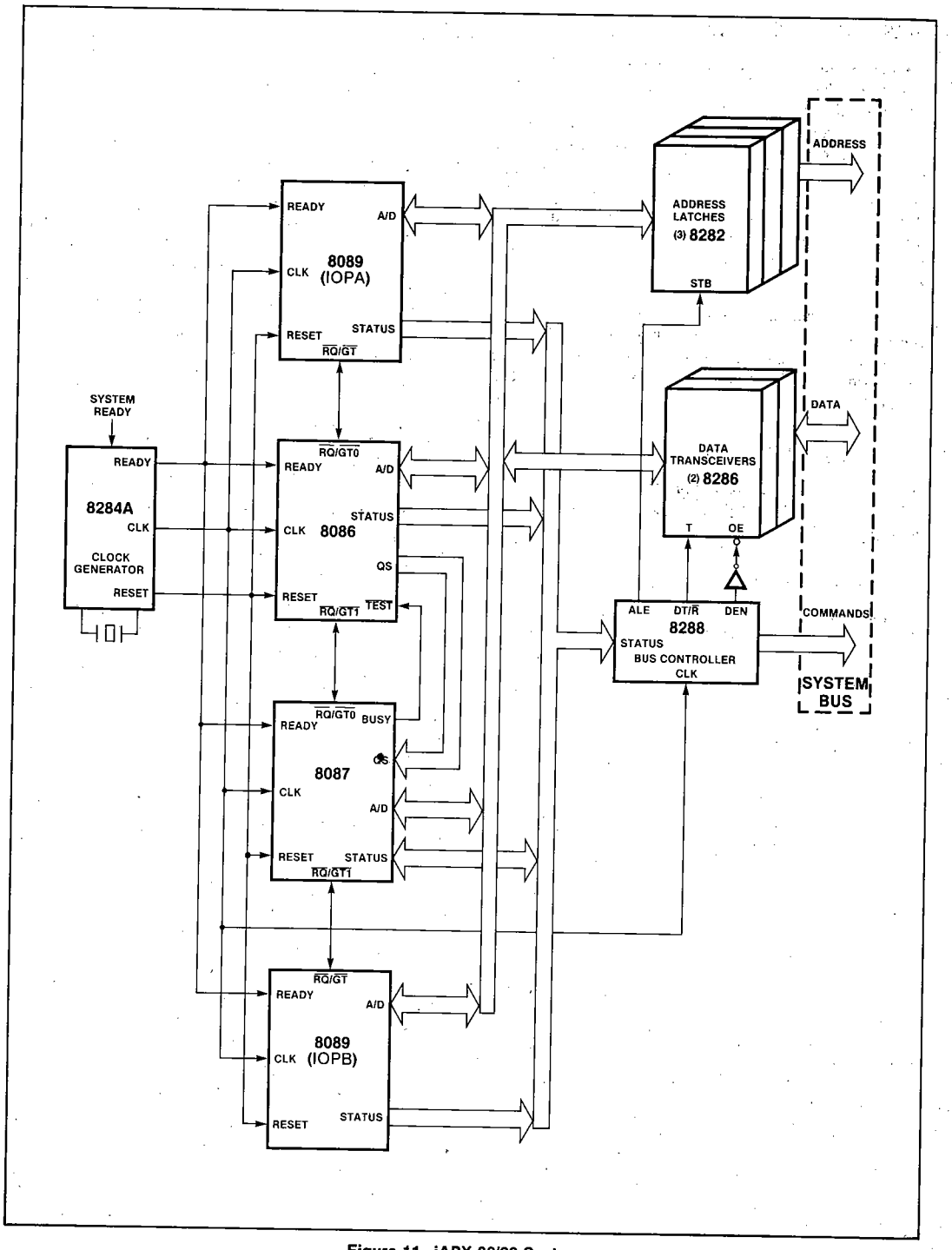


Figure 11. iAPX 86/22 System

iAPX 86/22, 88/22

An 86/22 system offers two alternates regarding to which IOP to connect an I/O device. Each IOP will offer a different maximum delay time to servide an I/O request. (See Fig. 11)

The second 8089 (IOPA) must use the RQ/GT0 pin of the host. With two IOPs the designer must decide which IOP services which I/O devices, determined by the maximum wait time allowed between when an I/O device requests IOP service and the IOP can respond. The maximum service delay times of the two IOPs can be very different. It makes little difference which of the two host RQ/GT pins are used.

The different wait times are due to the non-preemptive nature of bus grants between the two host RQ/GT pins. No communication of a need to use the local bus is possible between IOPA and the 8087/IOPB combination. Any request for the local bus by the IOPA must wait in the worst case for the host, 8087, and IOPB to finish their longest sequence of memory cycles. IOPB must wait in the worst case for the host and IOPA to finish their longest sequence of memory cycles. The 8087 has little effect on the maximum wait time of IOPB.

DELAY EFFECTS OF THE 8087

The delay effects of the 8087 on IOPA can be significant. When executing special instructions (FSAVE, FNSAVE, FRSTOR), the 8087 can perform 50 or 96 consecutive memory cycles with an 8086 or 8088 host, respectively. These instructions do not affect response time to local bus requests seen by an IOPB.

If the 8087 is performing a series of memory cycles while executing these instructions, and IOPB requests the local bus, the 8087 will stop its current memory activity, then release the local bus to IOPB.

The 8087 cannot release the bus to IOPA since it cannot know that IOPA wants to use the local bus, like it can for IOPB.

REDUCING 8087 DELAY EFFECTS

For 86/22 or 88/22 systems requiring lower maximum wait times for IOPA, it is possible to reduce the worst case bus usage of the 8087. If three 8087 instructions are never executed; namely FSAVE, FNSAVE, or FRSTOR, the maximum number of consecutive memory cycles performed by the 8087 is 10 or 16 for an 8086 or 8088 host respectively. The function of these instructions can be emulated with other 8087 instructions.

Appendix B shows an example of how these three instructions can be emulated. This improvment does have a cost, in the increased execution time of 427 or 747 additional clocks for an 8086 or 8088 respectively, for the equivalent save and restore operations. These operations appear in time-critical context-switching functions of an operating system or interrupt handler. This technique has no affect on the maximum wait time seen by IOPB or wait time seen by IOPA due to IOPB.

Which IOP to connect to which I/O device in an 86/22 or 88/22 system will depend on how quickly an I/O request by the device must be serviced by the IOP. This maximum time must be greater than the sum of the maximum delay of the IOP and the maximum wait time to gain control of the local bus by the IOP.

If neither IOP offers a fast enough response time, consider remote operation of the IOP.

8087 INT Connection

The next decision in adding the 8087 to an 8086 or 8088 system is where to attach the INT signal of the 8087. The INT pin of the 8087 provides an external indication of software-selected numeric errors. The numeric program will stop until something is done about the error. Deciding where to connect the INT signal can have important consequences on other interrupt handlers.

WHAT ARE NUMERIC ERRORS?

A numeric error occurs in the NPX whenever an operation is attempted with invalid operands or attempts to produce a result which cannot be represented. If an incorrect or questionable operation is attempted by a program, the NPX will always indicate the event. Examples of errors on the NPX are: 1/0, square root of -1, and reading from an empty register. For a detailed description of when the 8087 detects a numeric error, refer to the *Numerics Supplement*. (See Lit. Ref).

WHAT TO DO ABOUT NUMERIC ERRORS

Two possible courses of action are possible when a numeric error occurs. The NPX can itself handle the error, allowing numeric program execution to continue undisturbed, or software in the host can handle the error. To have the 8087 handle a numeric error, set its associated mask bit in the NPX control word. Each numeric error may be individually masked.

The NPX has a default fixup action defined for all possible numeric errors when they are masked. The default actions were carefully selected for their generality and safety.

For example, the default fixup for the precision error is to round the result using the rounding rules currently in effect. If the invalid error is masked, the NPX will generate a special value called indefinite as the result of any invalid operation.

NUMERIC ERRORS (CON'T)

Any arithmetic operation with an indefinite operand will always generate an indefinite result. In this manner, the result of the original invalid operation will propagate throughout the program wherever it is used.

When a questionable operation such as multiplying an unnormal value by a normal value occurs, the NPX will signal this occurrence by generating an unnormal result.

The required response by host software to a numeric error will depend on the application. The needs of each application must be understood when deciding on how to treat numeric errors. There are three attitudes towards a numeric error:

1) No response required. Let the NPX perform the default fixup.
2) Stop everything, something terrible has happened!
3) Oh, not again! But don't disrupt doing something more important.

SIMPLE ERROR HANDLING

Some very simple applications may mask all of the numeric errors. In this simple case, the 8087 INT signal may be left unconnected since the 8087 will never assert this signal. If any numeric errors are detected during the course of executing the program, the NPX will generate a safe result. It is sufficient to test the final results of the calculation to see if they are valid.

Special values like not-a-number (NAN), infinity, indefinite, denormals, and unnormals indicate the type and severity of earlier invalid or questionable operations.

SEVERE ERROR HANDLING

For dedicated applications, programs should not generate or use any invalid operands. Furthermore, all numbers should be in range. An operand or result outside this range indicates a severe fault in the system. This situation may arise due to invalid input values, program error, or hardware faults. The integrity of the program and hardware is in question, and immediate action is required.

In this case, the INT signal can be used to interrupt the program currently running. Such an interrupt would be of high priority. The interrupt handler responsible for numeric errors might perform system integrity tests and then restart the system at a known, safe state. The handler would not normally return to the point of error.

Unmasked numeric errors are very useful for testing programs. Correct use of synchronization, (Page 21), allows the programmer to find out exactly what operands, instruction, and memory values caused the error. Once testing has finished, an error then becomes much more serious.

The *8086 Family Numerics Supplement* recommends masking all errors except invalid. (See Lit. Ref.). In this case the NPX will safely handle such errors as underflow, overflow, or divide by zero. Only truly questionable operations will disturb the numerics program execution.

An example of how infinities and divide by zero can be harmless occurs when calculating the parallel resistance of several values with the standard formula (Figure 12). If R1 becomes zero, the circuit resistance becomes 0. With divide by zero and precision masked, the NPX will produce the correct result.

NUMERIC EXCEPTION HANDLING

For some applications, a numeric error may not indicate a severe problem. The numeric error can indicate that a hardware resource has been exhausted, and the software must provide more. These cases are called exceptions since they do not normally arise.

Special host software will handle numeric error exceptions when they infrequently occur. In these cases, numeric exceptions are expected to be recoverable although not requiring immediate service by the host. In effect, these exceptions extend the functionality of the NDP. Examples of extensions are: normalized only arithmetic, extending the register stack to memory, or tracing special data values.

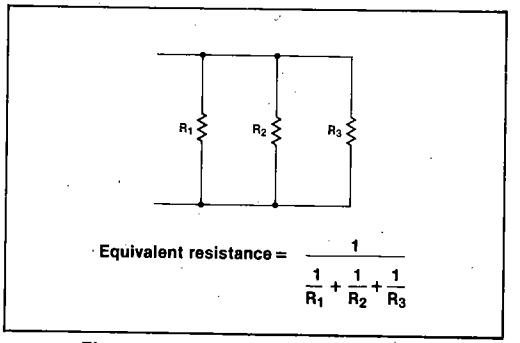


Figure 12. Infinity Arithmetic Example

HOST INTERRUPT OVERVIEW

The host has only two possible interrupt inputs, a non-maskable interrupt (NMI) and a maskable interrupt (INTR). Attaching the 8087 INT pin to the NMI input is not recommended. The following problems arise: NMI cannot be masked, it is usually reserved for more important functions like sanity timers or loss of power signal, and Intel supplied software for the NDP will not support NMI interrupts. The INTR input of the host allows interrupt masking in the CPU, using an Intel 8259A Programmable Interrupt Controller (PIC) to resolve multiple interrupts, and has Intel support.

NUMERIC INTERRUPT CHARACTERISTICS

Numeric error interrupts are different from regular instruction error interrupts like divide by zero. Numeric interrupts from the 8087 can occur long after the ESCAPE instruction that started the failing operation. For example, after starting a numeric multiply operation, the host may respond to an external interrupt and be in the process of servicing it when the 8087 detects an overflow error. In this case the interrupt is a result of some earlier, unrelated program.

From the point of view of the currently executing interrupt handler, numeric interrupts can come from only two sources: the current handler or a lower priority program.

To explicitly disable numeric interrupts, it is recommended that numeric interrupts be disabled at the 8087. The code example of Figure 13 shows how to disable any pending numeric interrupts then reenable them at the end of the handler. This code example can be safely placed in any routine which must prevent numeric interrupts from occurring. Note that the ESCAPE instructions act as NOPs if an 8087 is not present in the system. It is not recommended to use numeric mnemonics since they may be converted to emulator calls, which run comparatively slow, if the 8087 emulator used.

Interrupt systems have specific functions like fast response to external events or periodic execution of system routines. Adding an 8087 interrupt should not effect these functions. Desirable goals of any 8087 interrupt configuration are:

— Hide numeric interrupts from interrupt handlers that don't use the 8087. Since they didn't cause the numeric interrupt why should they be interrupted?

— Avoid adding code to interrupt handlers that don't use the 8087 to prevent interruption by the 8087.

— Allow other higher priority interrupts to be serviced while executing a numeric exception handler.

— Provide numeric exception handling for interrupt service routines which use the 8087.

— Avoid deadlock as described in a later section (page 24)

```
;
; Disable any possible numeric interrupt from the 8087. This code is safe to place in any
; procedure. If an 8087 is not present, the ESCAPE instructions will act as nops. These
; instructions are not affected by the TEST pin of the host. Using the 8087 emulator will not
; convert these instructions into interrupts. A word variable, called control, is required to hold
; the 8087 control word. Control must not be changed until it is reloaded into the 8087.
;
ESC   15, control          ; (FNSTCW) Save current 8087 control word
NOP                        ; Delay while 8087 saves current control
NOP                        ; register value
ESC   28,cx                ; (FNDISI) Disable any 8087 interrupts
                           ; Set IEM bit in 8087 control register
                           ; The contents of cx is irrelevant
                           ; Interrupts can now be enabled

                        (Your Code Here)

;
; Reenable any pending interrupts in the 8087. This instruction does not disturb any 8087 instruction
; currently in progress since all it does is change the IEM bit in the control register.
;
TEST  control, 80H         ; Look at IEM bit
JNZ   $+4                  ; If IEM = 1 skip FNENI
ESC   28,ax                ; (FNENI) reenable 8087 interrupts
```

Figure 13. Inhibit/Enable 8087 Interrupts

Recommended Interrupt Configurations

Five categories cover most uses of the 8087 interrupt in fixed priority interrupt systems. For each category, an interrupt configuration is suggested based on the goals mentioned above.

1. All errors on the 8087 are always masked. Numeric interrupts are not possible. Leave the 8087 INT signal unconnected.
2. The 8087 is the only interrupt in the system. Connect the 8087 INT signal directly to the host's INTR input. (See Figure 14 on page 19). A bus driver supplies interrupt vector 10_{16} for compatibility with Intel supplied software.
3. The 8087 interrupt is a stop everything event. Choose a high priority interrupt input that will terminate all numerics related activity. This is a special case since the interrupt handler may never return to the point of interruption (i.e. reset the system and restart rather than attempt to continue operation).
4. Numeric exceptions or numeric programming errors are expected and all interrupt handlers either don't use the 8087 or only use it with all errors masked. Use the lowest priority interrupt input. The 8087 interrupt handler should allow further interrupts by higher priority events. The PIC's priority system will automatically prevent the 8087 from disturbing other interrupts without adding extra code to them.
5. Case 4 holds except that interrupt handlers may also generate numeric interrupts. Connect the 8087 INT signal to multiple interrupt inputs. One input would still be the lowest priority input as in case 4. Interrupt handlers that may generate a numeric interrupt will require another 8087 INT connection to the next highest priority interrupt. Normally the higher priority numeric interrupt inputs would be masked and the low priority numeric interrupt enabled. The higher priority interrupt input would be unmasked only when servicing an interrupt which requires 8087 exception handling.

All of these configurations hide the 8087 from all interrupt handlers which do not use the 8087. Only those interrupt handlers that use the 8087 are required to perform any special 8087 related interrupt control activities.

A conflict can arise between the desired PIC interrupt input and the required interrupt vector of 10_{16} for compatibility with Intel software for numeric interrupts. A simple solution is to use more than one interrupt vector for numeric interrupts, all pointing at the same 8087 interrupt handler. Design the numeric interrupt handler such that it need not know what the interrupt vector was (i.e. don't use specific EOI commands).

If an interrupt system uses rotating interrupt priorities, it will not matter which interrupt input is used.

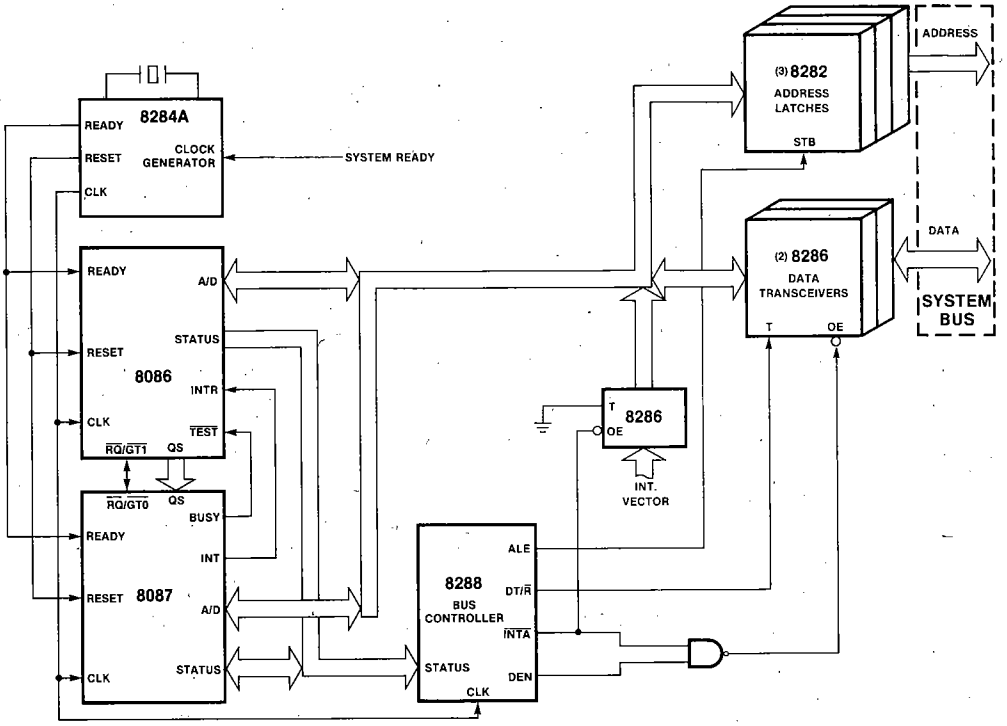


Figure 14. iAPX 86/20 With Numerics Interrupt Only

GETTING STARTED IN SOFTWARE

Now we are ready to run numeric programs. Developing numeric software will be a new experience to some programmers. This section of the application note is aimed at describing the programming environment and providing programming guidelines for the NPX. The term NPX is used to emphasize that no distinction is made between the 8087 component or an emulated 8087.

Two major areas of numeric software can be identified: systems software and applications software. Products such as iRMX™ 86 provide system software as an off-the-shelf product. Some applications use specially developed systems software optimized to their needs.

Whether the system software is specially tailored or common, they share issues such as using concurrency, maintaining synchronization between the host and 8087, and establishing programming conventions. Applications software directly performs the functions of the application. All applications will be concerned with initialization and general programming rules for the NPX. Systems software will be more concerned with context switching, use of the NPX by interrupt handlers, and numeric exception handlers.

How to Initialize the NPX

The first action required by the NPX is initialization. This places the NPX in a known state, unaffected by other activity performed earlier. This initialization is similar to that caused by the RESET signal of the 8087. All the error masks are set, all registers are tagged empty, the TOP field is set to 0, default rounding, precision, and infinity controls are set. The 8087 emulator requires more initialization than the component. Before the emulator may be used, all its interrupt vectors must be set to point to the correct entry points within the emulator.

To provide compatibility between the emulator and component in this special case, a call to an external procedure should be used before the first numeric instruction. In ASM86 the programmer must call the external function INIT87. (Fig. 15). For PLM86, the programmer must call the built-in function INIT$REAL$MATH$UNIT. PLM86 will call INIT87 when executing the INIT$REAL$MATH$UNIT built-in function.

The function supplied for INIT87 will be different, depending on whether the emulator library, called E8087.LIB, or component library, called 8087.LIB, were used at link time. INIT87 will execute either an FNINIT instruction for the 8087 or initialize the 8087 emulator interrupt vectors, as appropriate.

Concurrency Overview

With the NPX initialized, the next step in writing a numeric program is learning about concurrent execution within the NDP.

Concurrency is a special feature of the 8087, allowing it and the host to simultaneously execute different instructions. The 8087 emulator does not provide concurrency since it is implemented by the host.

The benefit of concurrency to an application is higher performance. All Intel high level languages automatically provide for and manage concurrency in the NDP. However, in exchange for the added performance, the assembly language programmer must understand and manage some areas of concurrency. This section is for the assembly language programmer or well-informed, high level language programmer.

Whether the 8087 emulator or component is used, care should be taken by the assembly language programmer to follow the rules described below regarding synchronization. Otherwise, the program may not function correctly with current or future alternatives for implementing the NDP.

Concurrency is possible in the NDP because both the host and 8087 have separate arithmetic and control units. The host and coprocessor automatically decide who will perform any single instruction. The existence of the 8087 as a separate unit is not normally apparent.

Numeric instructions, which will be executed by the 8087, are simply placed in line with the instructions for the host. Numeric instructions are executed in the same order as they are encountered by the host in its instruction stream. Since operations performed by the 8087 generally require more time than operations performed by the host, the host can execute several of its instructions while the 8087 performs one numeric operation.

```
IN PLM86:
    CALL INIT$REAL$MATH$UNIT;

IN ASM86:
        EXTRN                   INIT87:FAR
        •
        •
        •
        •
        CALL                    INIT87
```

Figure 15. 8087 Initialization

MANAGING CONCURRENCY

Concurrent execution of the host and 8087 is easy to establish and maintain. The activities of numeric programs can be split into two major areas: program control and arithmetic. The program control part performs activities like deciding what functions to perform, calculating addresses of numeric operands, and loop control. The arithmetic part simply performs the adds, subtracts, multiplies, and other operations on the numeric operands. The NPX and host are designed to handle these two parts separately and efficiently.

Managing concurrency is necessary because the arithmetic and control areas must converge to a well-defined state when starting another numeric operation. A well-defined state means all previous arithmetic and control operations are complete and valid.

Normally, the host waits for the 8087 to finish the current numeric operation before starting another. This waiting is called synchronization.

Managing concurrent execution of the 8087 involves three types of synchronization: instruction, data, and error. Instruction and error synchronization are automatically provided by the compiler or assembler. Data synchronization must be provided by the assembly language progammer or compiler.

Instruction Synchronization

Instruction synchronization is required because the 8087 can only perform one numeric operation at a time. Before any numeric operation is started, the 8087 must have completed all activity from previous instructions.

The WAIT instruction on the host lets it wait for the 8087 to finish all numeric activity before starting another numeric instruction. The assembler automatically provides for instruction synchronization since a WAIT instruction is part of most numeric instructions. A WAIT instruction requires 1 byte code space and 2.5 clocks average execution time overhead.

Instruction synchronization as provided by the assembler or a compiler allows concurrent operation in the NDP. An execution time comparison of NDP concurrency and non-concurrency is illustrated in Figure 16. The non-concurrent program places a WAIT instruction immediately after a multiply instruction ESCAPE instruction. The 8087 must complete the multiply operation before the host executes the MOV instruction on statement 2. In contrast, the concurrent example allows the host to calculate the effective address of the next operand while the 8087 performs the multiply. The execution time of the concurrent technique is the longest of the host's execution time from line 2 to 5 and the execution time of the 8087 for a multiply instruction. The execution time of the non-concurrent example is the sum of the execution times of statements 1 to 5.

```
;
; This code macro defines two instructions which do not allow any concurrency of execution with
; the host. A register version and memory version of the instruction is shown. It is assumed that the
; 8087 is always idle from the previous instruction. Allow space for emulator fixups.
;
R233 Record RF6:2, Mid3:3, RF7:3
CodeMacro NCMUL dst:T, src:F
RNfix 000B
R233 (11B, 001B, src)
RWfix
EndM

CodeMacro NCMUL memop:Mq
RNfixM 100B, memop
ModRM 001B, memop
RWfix
EndM
```

Statement	Concurrent		Non Concurrent	
1	FMUL	st(0), st(1)	NCMUL	st(0), st(1)
2	MOV	ax, size A	MOV	ax, size A
3	MUL	index	MUL	index
4	MOV	bx, ax	MOV	bx, ax
5	FMUL	A [bx]	NCMUL	A [bx]

Figure 16. Concurrent Versus Non-Concurrent Program

Data Synchronization

Managing concurrency requires synchronizing data references by the host and 8087.

Figure 17 shows four possible cases of the host and 8087 sharing a memory value. The second two cases require the FWAIT instruction shown for data synchronization. In the first two cases, the host will finish with the operand I before the 8087 can reference it. The coprocessor interface guarantees this. In the second two cases, the host must wait for the 8087 to finish with the memory operand before proceeding to reuse it. The FWAIT instruction in case 3 forces the host to wait for the 8087 to read I before changing it. In case 4, the FWAIT prevents the host from reading I before the 8087 sets its value.

Obviously, the programmer must recognize any form of the two cases shown above which require explicit data synchronization. Data synchronization is not a concern when the host and 8087 are using different memory operands during the course of one numeric instruction. Figure 16 shows such an example of the host performing activity unrelated to the current numeric instruction being executed by the 8087. Correct recognition of these cases by the programmer is the price to be paid for providing concurrency at the assembly language level.

Automatic Data Synchronization

Two methods exist to avoid the need for manual recognition of when data synchronization is needed: use a high level language which will automatically establish concurrency and manage it, or sacrifice some performance for automatic data synchronization by the assembler.

When a high level language is not adequate, the assembler can be changed to always place a WAIT instruction after the ESCAPE instruction. Figure 18 shows an example of how to change the ASM86 code macro for the FIST instruction to automatically place an FWAIT instruction after the ESCAPE instruction. The lack of any possible concurrent execution between the host and 8087 while the FIST instruction is executing is the price paid for automatic data synchronization.

An explicit FWAIT instruction for data synchronization, can be eliminated by using a subsequent numeric instruction. After this subsequent instruction has started execution, all memory references in earlier numeric instructions are complete. Reaching the next host instruction after the synchronizing numeric instruction indicates previous numeric operands in memory are available.

The data synchronization purpose of any FWAIT or numeric instruction must be well documented. Otherwise, a change to the program at a later time may remove the synchronizing numeric instruction, causing program failure, as:

```
FISTP   I
FMUL
MOV     AX, I     ; I is safe to use
```

```
Case 1:
    MOV   I, 1
    FILD  I

Case 2:
    MOV   AX, I
    FISTP I

Case 3:
    FILD  I
    FWAIT
    MOV   I,5

Case 4:
    FISTP I
    FWAIT
    MOV   AX,I
```

Figure 17. Data Exchange Example

```
;
; This is a code macro to redefine the FIST
; instruction to prevent any concurrency
; while the instruction runs. A wait
; instruction is placed immediately after the
; escape to ensure the store is done
; before the program may continue. This
; code macro will work with the 8087
; emulator, automatically replacing the
; wait escape with a nop.
;
CodeMacro FIST memop: Mw
RfixM 111B, memop
ModRM 010B, memop
RWfix
EndM
```

Figure 18. Non-Concurrent FIST Instruction Code Macro

DATA SYNCHRONIZATION RULES EXCEPTIONS

There are five exceptions to the above rules for data synchronization. The 8087 automatically provides data synchronization for these cases. They are necessary to avoid deadlock (described on page 24). The instructions FSTSW/FNSTSW, FSTCW/FNSTCW, FLDCW, FRSTOR, and FLDENV do not require any waiting by the host before it may read or modify the referenced memory location.

The 8087 provides the data synchronization by preventing the host from gaining control of the local bus while these instructions execute. If the host cannot gain control of the local bus, it cannot change a value before the 8087 reads it, or read a value before the 8087 writes into it.

The coprocessor interface guarantees that, when the host executes one of these instructions, the 8087 will immediately request the local bus from the host. This request is timed such that, when the host finishes the read operation identifying the memory operand, it will always grant the local bus to the 8087 before the host may use the local bus for a data reference while executing a subsequent instruction. The 8087 will not release the local bus to the host until it has finished executing the numeric instruction.

Error Synchronization

Numeric errors can occur on almost any numeric instruction at any time during its execution. Page 15 describes how a numeric error may have many interpretations, depending on the application. Since the response to a numeric error will depend on the application, this section covers topics common to all uses of the NPX. We will review why error synchronization is needed and how it is provided.

Concurrent execution of the host and 8087 requires synchronization for errors just like data references and numeric instructions. In fact, the synchronization required for data and instructions automatically provides error synchronization.

However, incorrect data or instruction synchronization may not cause a problem until a numeric error occurs. A further complication is that a programmer may not expect his numeric program to cause numeric errors, but in some systems they may regularly happen. To better understand these points, let's look at what can happen when the NPX detects an error.

ERROR SYNCHRONIZATION FOR EXTENSIONS

The NPX can provide a default fixup for all numeric errors. A program can mask each individual error type to indicate that the NPX should generate a safe, reasonable result. The default error fixup activity is simply treated as part of the instruction which caused the error. No external indication of the error will be given. A flag in the numeric status register will be set to indicate that an error was detected, but no information regarding where or when will be available.

If the NPX performs its default action for all errors, then error synchronization is never exercised. But this is no reason to ignore error synchronization.

Another alternative exists to the NPX default fixup of an error. If the default NPX response to numeric errors is not desired, the host can implement any form of recovery desired for any numeric error detectable by the NPX. When a numeric error is unmasked, and the error occurs, the NPX will stop further execution of the numeric instruction. The 8087 will signal this event on the INT pin, while the 8087 emulator will cause interrupt 10_{16} to occur. The 8087 INT signal is normally connected to the host's interrupt system. Refer to page 18 for further discussion on wiring the 8087 INT pin.

Interrupting the host is a request from the NPX for help. The fact that the error was unmasked indicates that further numeric program execution under the arithmetic and programming rules of the NPX is unreasonable. Error synchronization serves to insure the NDP is in a well defined state after an unmasked numeric error occured. Without a well defined state, it is impossible to figure out why the error occured.

Allowing a correct analysis of the error is the heart of error synchronization.

NDP ERROR STATES

If concurrent execution is allowed, the state of the host when it recognizes the interrupt is undefined. The host may have changed many of its internal registers and be executing a totally different program by the time it is interrupted. To handle this situation, the NPX has special registers updated at the start of each numeric instruction to describe the state of the numeric program when the failed instruction was attempted. (See Lit. Ref. p. iii)

Besides programmer comfort, a well-defined state is important for error recovery routines. They can change the arithmetic and programming rules of the 8087. These changes may redefine the default fixup from an error, change the appearance of the NPX to the programmer, or change how arithmetic is defined on the NPX.

EXTENSION EXAMPLES

A change to an error response might be to automatically normalize all denormals loaded from memory. A change in appearance might be extending the register stack to memory to provide an "infinite" number of numeric registers. The arithmetic of the 8087 can be changed to automatically extend the precision and range of variables when exceeded. All these functions can be implemented on the NPX via numeric errors and associated recovery routines in a manner transparent to the programmer.

Without correct error synchronization, numeric subroutines will not work correctly in the above situations.

Incorrect Error Synchronization

An example of how some instructions written without error synchronization will work initially, but fail when moved into a new environment is:

```
FILD    COUNT
INC     COUNT
FSQRT
```

Three instructions are shown to load an integer, calculate its square root, then increment the integer. The coprocessor interface of the 8087 and synchronous execution of the 8087 emulator will allow this program to execute correctly when no errors occur on the FILD instruction.

But, this situation changes if the numeric register stack is extended to memory on an 8087. To extend the NPX stack to memory, the invalid error is unmasked. A push to a full register or pop from an empty register will cause an invalid error. The recovery routine for the error must recognize this situation, fixup the stack, then perform the original operation.

The recovery routine will not work correctly in the example. The problem is that there is no guarantee that COUNT will not be incremented before the 8087 can interrupt the host. If COUNT is incremented before the interrupt, the recovery routine will load a value of COUNT one too large, probably causing the program to fail.

Error Synchronization and WAITs

Error synchronization relies on the WAIT instructions required by instruction and data synchronization and the INT and BUSY signals of the 8087. When an unmasked error occurs in the 8087, it asserts the BUSY and INT signals. The INT signal is to interrupt the host, while the BUSY signal prevents the host from destroying the current numeric context.

The BUSY signal will never go inactive during a numeric instruction which asserts INT.

The WAIT instructions supplied for instruction synchronization prevent the host from starting another numeric instruction until the current error is serviced. In a like manner, the WAIT instructions required for data synchronization prevent the host from prematurely reading a value not yet stored by the 8087, or overwriting a value not yet read by the 8087.

The host has two responsibilities when handling numeric errors.1.) It must not disturb the numeric context when an error is detected, and 2.) it must clear the numeric error and attempt recovery from the error. The recovery program invoked by the numeric error may resume program execution after proper fixup, display the state of the NDP for programmer action, or simply abort the program. In any case, the host must do something with the 8087. With the INT and BUSY signals active, the 8087 cannot perform any useful work. Special instructions exist for controlling the 8087 when in this state. Later, an example is given of how to save the state of the NPX with an error pending. (See page 29)

Deadlock

An undesirable situation may result if the host cannot be interrupted by the 8087 when asserting INT. This situation, called deadlock, occurs if the interrupt path from the 8087 to the host is broken.

The 8087 BUSY signal prevents the host from executing further instructions (for instruction or data synchronization) while the 8087 waits for the host to service the exception. The host is waiting for the 8087 to finish the current numeric operation. Both the host and 8087 are waiting on each other. This situation is stable unless the host is interrupted by some other event.

Deadlock has varying affects on the NDP's performance. If no other interrupts in the system are possible, the NDP will wait forever. If other interrupts can arise, then the NDP can perform other functions, but the affected numeric program will remain "frozen".

SOLVING DEADLOCK

Finding the break in the interrupt path is simple. Look for disabled interrupts in the following places: masked interrupt enable in the host, explicitly masked interrupt request in the interrupt controller, implicitly masked interrupt request in the interrupt controller due to a higher priority interrupt in service, or other gate functions, usually in TTL, on the host interrupt signal.

DEADLOCK AVOIDANCE

Application programmers should not be concerned with deadlock. Normally, applications programs run with unmasked numeric errors able to interrupt them. Deadlock is not possible in this case. Traditionally, systems software or interrupt handlers may run with numeric interrupts disabled. Deadlock prevention lies in this domain. The golden rule to abide by is: "Never wait on the 8087 if an unmasked error is possible and the 8087 interrupt path may be broken."

Error Synchronization Summary

In summary, error synchronization involves protecting the state of the 8087 after an exception. Although not all applications may initially require error synchronization, it is just good programming practice to follow the rules. The advantage of being a "good" numerics programmer is generality of your program so it can work in other, more general environments.

Summary

Synchronization is the price for concurrency in the NDP. Intel high level language compilers will automatically provide concurrency and manage it with synchronization. The assembly language programmer can choose between using concurrency or not. Placing a WAIT instruction immediately after any numeric instruction will prevent concurrency and avoid synchronization concerns.

The rules given above are complete and allow concurrency to be used to full advantage.

Synchronization and the Emulator

The above discussion on synchronization takes on special meaning with the 8087 emulator. The 8087 emulator does not allow any concurrency. All numeric operand memory references, error tests, and wait for instruction completion occur within the emulator. As a result, programs which do not provide proper instruction, data, or error synchronization may work with the 8087 emulator while failing on the component.

Correct programs for the 8087 work correctly on the emulator.

Special Control Instructions of the NPX

The special control instructions of the NPX: FNINIT, FNSAVE, FNSTENV, FRSTOR, FLDENV, FLDCW, FNSTSW, FNSTCW, FNCLEX, FNENI, and FNDISI remove some of the synchronization requirements mentioned earlier. They are discussed here since they represent exceptions to the rules mentioned on page 21.

The instructions FNINIT, FNSAVE, FNSTENV, FNSTSW, FNCLEX, FNENI, and FNDISI do not wait for the current numeric instruction to finish before they execute. Of these instructions, FNINIT, FNSTSW, FNCLEX, FNENI and FNDISI will produce different results, depending on when they are executed relative to the current numeric instruction.

For example, FNCLEX will cause a different status value to result from a concurrent arithmetic operation, depending on whether is is executed before or after the error status bits are updated at the end of the arithmetic operation. The intended use of FNCLEX is to clear a known error status bit which has caused BUSY to be asserted, avoiding deadlock.

FNSTSW will safely, without deadlock, report the busy and error status of the NPX independent of the NDP interrupt status.

FNINIT, FNENI, and FNDISI are used to place the NPX into a known state independent of its current state. FNDISI will prevent an unmasked error from asserting BUSY without disturbing the current error status bits. Appendix A shows an example of using FNDISI.

The instructions FNSAVE and FNSTENV provide special functions. They allow saving the state of the NPX in a single instruction when host interrupts are disabled.

Several host and numeric instructions are necessary to save the NPX status if the interrupt status of the host is unknown. Appendix A and B show examples of saving the NPX state. As the *Numerics Supplement* explains, host interrupts must always be disabled when executing FNSAVE or FNSTENV.

The seven instructions FSTSW/FNSTSW, FSTCW/FNSTCW, FLDCW, FLDENV, and FRSTOR do not require explicit WAIT instructions for data synchronization. All of these instructions are used to interrogate or control the numeric context.

Data synchronization for these instructions is automatically provided by the coprocessor interface. The 8087 will take exclusive control of the memory bus, preventing the host from interfering with the data values before the 8087 can read them. Eliminating the need for a WAIT instruction avoids potential deadlock problems.

The three load instructions FLDCW, FLDENV, and FRSTOR can unmask a numeric error, activating the 8087 BUSY signal. Such an error was the result of a previous numeric instruction and is not related to any fault in the instruction.

Data synchronization is automatically provided since the host's interrupts are usually disabled in context switching or interrupt handling, deadlock might result if the host executed a WAIT instruction with its interrupts disabled after these instructions. After the host interrupts are enabled, an interrupt will occur if an unmasked error was pending.

PROGRAMMING TECHNIQUES

The NPX provides a stack-oriented register set with stack-oriented instructions for numeric operands. These registers and instructions are optimized for numeric programs. For many programmers, these are new resources with new programming options available.

Using Numeric Registers and Instructions

The register and instruction set of the NDP is optimized for the needs of numeric and general purpose programs. The host CPU provides the instructions and data types needed for general purpose data processing, while the 8087 provides the data types and instructions for numeric processing.

The instructions and data types recognized by the 8087 are different from the CPU because numeric program requirements are different from those of general purpose programs. Numeric programs have long arithmetic expressions where a few temporary values are used in a few statements. Within these statements, a single value may be referenced many times. Due to the time involved to transfer values between registers and memory, a significant speed optimization is possible by keeping numbers in the NPX register file.

In contrast, a general data processor is more concerned with addressing data in simple expressions and testing the results. Temporary values, constant across several instructions, are not as common nor is the penalty as large for placing them in memory.As a result it is simpler for compilers and programmers to manage memory based values.

NPX Register Usage

The eight numeric registers in the NDP are stack oriented. All numeric registers are addressed relative to a value called the TOP pointer, defined in the NDP status register. A register address given in an instruction is added to the TOP value to form the internal absolute address. Relative addressing of numeric registers has advantages analogous to those of relative addressing of memory operands.

Two modes are available for addressing the numeric registers. The first mode implicitly uses the top and optional next element on the stack for operands. This mode does not require any addressing bits in a numeric instruction. Special purpose instructions use this mode since full addressing flexibility is not required.

The other addressing mode allows any other stack element to be used together with the top of stack register. The top of stack or the other register may be specified as the destination. Most two-operand arithmetic instructions allow this addressing mode. Short, easy to develop numeric programs are the result.

Just as relative addressing of memory operands avoids concerns with memory allocation in other parts of a program, top relative register addressing allows registers to be used without regard for numeric register assignments in other parts of the program.

STACK RELATIVE ADDRESSING EXAMPLE

Consider an example of a main program calling a subroutine, each using register addressing independent of the other. (Fig. 19) By using different values of the TOP field, different software can use the same relative register addresses as other parts of the program, but refer to different physical registers.

```
MAIN_PROGRAM:
  FLD       A
  FADD      ST, ST(1)
  CALL      SUBROUTINE      ; Argument is in ST(0)
  FSTP      B

SUBROUTINE:
  FLD       ST              ; ST(0) = ST(1) = Argument
  FSQRT                     : Main program ST(1) is
  FADD      C               ; safe in ST(2) here
  FMULP     ST(1), ST
  RET
```

Figure 19. Stack Relative Addressing Example

Of course, there is a limit to any physical resource. The NDP has eight numeric registers. Normally, programmers must ensure a maximum of eight values are pushed on the numeric register stack at any time. For time-critical inner loops of real-time applications, eight registers should contain all the values needed.

REGISTER STACK EXTENSION

This hardware limitation can be hidden by software. Software can provide "virtual" numeric registers, expanding the register stack size to 6000 or more.

The numeric register stack can be extended into memory via unmasked numeric invalid errors which cause an interrupt on stack overflow or underflow. The interrupt handler for the invalid error would manage a memory image of the numeric stack copying values into and out of memory as needed.

The NPX will contain all the necessary information to identify the error, failing instruction, required registers, and destination register. After correcting for the missing hardware resource, the original numeric operation could be repeated. Either the original numeric instruction could be single stepped or the affect of the instruction emulated by a composite of table-based numeric instructions executed by the error handler.

With proper data, error, and instruction synchronization, the activity of the error handler will be transparent to programs. This type of extension to the NDP allows programs to push and pop numeric registers without regard for their usage by other subroutines.

Programming Conventions

With a better understanding of the stack registers, let's consider some useful programming conventions. Following these conventions ensures compatibility with Intel support software and high level language calling conventions.

1) If the numeric registers are not extended to memory, the programmer must ensure that the number of temporary values left in the NPX stack and those registers used by the caller does not exceed 8. Values can be stored to memory to provide enough free NPX registers.
2) Pass the first seven numeric parameters to a subroutine in the numeric stack registers. Any extra parameters can be passed on the host's stack. Push the values on the register or memory stack in left to right order. If the subroutine does not need to allocate any more numeric registers, it can execute solely out of the numeric register stack. The eighth register can be used for arithmetic operations. All parameters should be popped off when the subroutine completes.
3) Return all numeric values on the numeric stack. The caller may now take advantage of the extended precision and flexible store modes of the NDP.
4) Finish all memory reads or writes by the NPX before exiting any subroutine. This guarantees correct data and error synchronization. A numeric operation based solely on register contents is safe to leave running on subroutine exit.
5) The operating mode of the NDP should be transparent across any subroutine. The operating mode is defined by the control word of the NDP. If the subroutine needs to use a different numeric operating mode than that of the caller, the subroutine should first save the current control word, set the new operating mode, then restore the original control word when completed.

PROGRAMMING EXAMPLES

The last section of this application note will discuss five programming examples. These examples were picked to illustrate NDP programming techniques and commonly used functions. All have been coded, assembled, and tested. However, no guarantees are made regarding their correctness.

The programming examples are: saving numeric context switching, save numeric context without FSAVE/FNSAVE, converting ASCII to floating point, converting floating point to ASCII, and trigonometric functions. Each example is listed in a different appendix with a detailed written description in the following text. The source code is available in machine readable form from the Intel Insite User's Library, "Interactive 8087 Instruction Interpreter," catalog item AA20.

The examples provide some basic functions needed to get started with the numeric data processor. They work with either the 8087 or the 8087 emulator with no source changes.

The context switching examples are needed for operating systems or interrupt handlers which may use numeric instructions and operands. Converting between floating point and decimal ASCII will be needed to input or output numbers in easy to read form. The trigonometric examples help you get started with sine or cosine functions and can serve as a basis for optimizations if the angle arguments always fall into a restricted range.

APPENDIX A

OVERVIEW

Appendix A shows deadlock-free examples of numeric context switching. Numeric context switching is required by interrupt handlers which use the NPX and operating system context switchers. Context switching consists of two basic functions, save the numeric context and restore it. These functions must work independent of the current state of the NPX.

Two versions of the context save function are shown. They use different versions of the save context instruction. The FNSAVE/FSAVE instructions do all the work of saving the numeric context. The state of host interrupts will decide which instruction to use.

Using FNSAVE

The FNSAVE instruction is intended to save the NPX context when host interrupts are disabled. The host does not have to wait for the 8087 to finish its current operation before starting this operation. Eliminating the instruction synchronization wait avoids any potential deadlock.

The 8087 Bus Interface Unit (BIU) will save this instruction when encountered by the host and hold it until the 8087 Floating point Execution Unit (FEU) finishes its current operation. When the FEU becomes idle, the BIU will start the FEU executing the save context operation.

The host can execute other non-numeric instructions after the FNSAVE while the BIU waits for the FEU to finish its current operation. The code starting at NO_INT_NPX_SAVE shows how to use the FNSAVE instruction.

When executing the FNSAVE instruction, host interrupts must be disabled to avoid recursions of the instruction. The 8087 BIU can hold only one FNSAVE instruction at a time. If host interrupts were not disabled, another host interrupt might cause a second FNSAVE instruction to be executed, destroying the previous one saved in the 8087 BIU.

It is not recommended to explicitly disable host interrupts just to execute an FNSAVE instruction. In general, such an operation may not be the best course of action or even be allowed.

If host interrupts are enabled during the NPX context save function, it is recommended to use the FSAVE instruction as shown by the code starting at NPX_SAVE. This example will always work, free of deadlock, independent of the NDP interrupt state.

Using FSAVE

The FSAVE instruction performs the same operation as FNSAVE but it uses standard instruction synchronization. The host will wait for the FEU to be idle before initiating the save operation. Since the host ignores all interrupts between completing a WAIT instruction and starting the following ESCAPE instruction, the FEU is ready to immediately accept the operation (since it is not signalling BUSY). No recursion of the save context operation in the BIU is possible. However, deadlock must be considered since the host executes a WAIT instruction.

To avoid deadlock when using the FSAVE instruction, the 8087 must be prevented from signalling BUSY when an unmasked error exists.

The Interrupt Enable Mask (IEM) bit in the NPX control word provides this function. When IEM = 1, the 8087 will not signal BUSY or INT if an unmasked error exists. The NPX instruction FNDISI will set the IEM independent of any pending errors without causing deadlock or any other errors. Using the FNDISI and FSAVE instructions together with a few other glue instructions allows a general NPX context save function.

Standard data and instruction synchronization is required after executing the FNSAVE/FSAVE instruction. The wait instruction following an FNSAVE/FSAVE instruction is always safe since all NPX errors will be masked as part of the instruction execution. Deadlock is not possible since the 8087 will eventually signal not busy, allowing the host to continue on.

PLACING THE SAVE CONTEXT FUNCTION

Deciding on where to save the NPX context in an interrupt handler or context switcher is dependent on whether interrupts can be enabled inside the function. Since interrupt latency is measured in terms of the maximum time interrupts are disabled, the maximum wait time of the host at the data synchronizing wait instruction after the FNSAVE or the FSAVE instruction is important if host interrupts are disabled while waiting.

The wait time will be the maximum single instruction execution time of the 8087 plus the execution time of the save operation. This maximum time will be approximately 1300 or 1500 clocks, depending on whether the host is an 8086 or 8088, respectively. The actual time will depend on how much concurrency of execution between the host and 8087 is provided. The greater the concurrency, the lesser the maximum wait time will be.

If host interrupts can be enabled during the context save function, it is recommended to use the FSAVE instruction for saving the numeric context in the interruptable section. The FSAVE instruction allows instruction and data synchronizing waits to be interruptable. This technique removes the maximum execution time of 8087 instructions from system interrupt latency time considerations.

It is recommended to delay starting the numeric save function as long as possible to maintain the maximum amount of concurrent execution between the host and the 8087.

Using FRSTOR

Restoring the numeric context with FRSTOR does not require a data synchronizing wait afterwards since the 8087 automatically prevents the host from interfering with the memory load operation.

The code starting with NPX_RESTORE illustrates the restore operation. Error synchronization is not necessary since the FRSTOR instruction itself does not cause errors, but the previous state of the NPX may indicate an error.

If further numeric instructions are executed after the FRSTOR, and the error state of the new NPX context is unknown, deadlock may occur if numeric exceptions cannot interrupt the host.

NPX_save

```
;
; General purpose save of NPX context. This function will work independent of the interrupt state of
; the NDP. Deadlock can not occur. 47 words of memory are required by the variable save_area.
; Register ax is not transparent across this code.
;
NPX_save:
    FNSTCW    save_area        ; Save IEM bit status
    NOP                        ; Delay while 8087 saves control register
    FNDISI                     ; Disable 8087 BUSY signal
    MOV       ax, save_area    ; Get original control word
    FSAVE     save_area        ; Save NPX context, the host can be safely interrupted while
                               ; waiting for the 8087 to finish. Deadlock is not possible since
    FWAIT                      ; IEM = 1.Wait for save to finish. Put original control word into
    MOV       save_area, ax    ; NPX context area. All done
```

no_int_NPX_save

```
;
; Save the NPX context with host interrupts disabled. No deadlock is possible. 47 words of memory
; are required by the variable save_area.
;
no_int_NPX_save:
    FNSAVE    save_area        ; Save NPX context. Wait for save to finish, no deadlock
    FWAIT                      ; is possible. Interrupts may be enabled now, all done
```

NPX_restore

```
;
; Restore the NPX context saved earlier. No deadlock is possible if no further numeric instructions
; are executed until the 8087 numeric error interrupt is enabled. The variable save_area is assumed
; to hold an NPX context saved earlier. It must be 47 words long.
;
NPX_restore:
    FRSTOR    save_area        ; Load new NPX context
```

APPENDIX B

OVERVIEW

Appendix B shows alternative techniques for switching the numeric context without using the FSAVE/FNSAVE or FRSTOR instructions. These alternative techniques are slower than those of Appendix A but they reduce the worst case continuous local bus usage of the 8087.

Only an iAPX 86/22 or iAPX 88/22 could derive any benefit from this alternative. By replacing all FSAVE/FNSAVE instructions in the system, the worst case local bus usage of the 8087 will be 10 or 16 consecutive memory cycles for an 8086 or 8088 host, respectively.

Instead of saving and loading the entire numeric context in one long series of memory transfers, these routines use the FSTENV/FNSTENV/FLDENV instructions and separate numeric register load/store instructions. Using separate load/store instructions for the numeric registers forces the 8087 to release the local bus after each numeric load/store instruction. The longest series of back-to-back memory transfers required by these instructions are 8/12 memory cycles for an 8086 or 8088 host, respectively. In contrast, the FSAVE/FNSAVE/FRSTOR instructions perform 50/94 back-to-back memory cycles for an 8086 or 8088 host.

Compatibility With FSAVE/FNSAVE

This function produces a context area of the same format produced by FSAVE/FNSAVE instructions. Other software modules expecting such a format will not be affected. All the same interrupt and deadlock considerations of FSAVE and FNSAVE also apply to FSTENV and FNSTENV. Except for the fact that the numeric environment is 7 words rather than the 47 words of the numeric context, all the discussion of Appendix A also applies here.

The state of the NPX registers must be saved in memory in the same format as the FSAVE/FNSAVE instructions. The program example starting at the label SMALL_BLOCK_NPX_SAVE illustrates a software loop that will store their contents into memory in the same top relative order as that of FSAVE/FNSAVE.

To save the registers with FSTP instructions, they must be tagged valid, zero, or special. This function will force all the registers to be tagged valid, independent of their contents or old tag, and then save them. No problems will arise if the tag value conflicts with the register's content for the FSTP instruction. Saving empty registers insures compatibility with the FSAVE/FNSAVE instructions. After saving all the numeric registers, they will all be tagged empty, the same as if an FSAVE/FNSAVE instruction had been executed.

Compatibility With FRSTOR

Restoring the numeric context reverses the procedure described above, as shown by the code starting at SMALL_BLOCK_NPX_RESTORE. All eight registers are reloaded in the reverse order. With each register load, a tag value will be assigned to each register. The tags assigned by the register load does not matter since the tag word will be overwritten when the environment is reloaded later with FLDENV.

Two assumptions are required for correct operation of the restore function: all numeric registers must be empty and the TOP field must be the same as that in the context being restored. These assumptions will be satisfied if a matched set of pushes and pops were performed between saving the numeric context and reloading it.

If these assumptions cannot be met, then the code example starting at NPX_CLEAN shows how to force all the NPX registers empty and set the TOP field of the status word.

small_block_NPX_save

```
;
; Save the NPX context independent of NDP interrupt state. Avoid using the FSAVE instruction to
; limit the worst case memory bus usage of the 8087. The NPX context area formed will appear the
; same as if an FSAVE instruction had written into it. The variable save_area will hold the NPX
; context and must be 47 words long. The registers ax, bx, and cx will not be transparent.
;
small_block_NPX_save:
    FNSTCW  save_area          ; Save current IEM bit
    NOP                        ; Delay while 8087 saves control register
    FNDISI                     ; Disable 8087 BUSY signal
    MOV     ax, save_area      ; Get original control word
    MOV     cx, 8              ; Set numeric register count
    XOR     bx, bx             ; Tag field value for stamping all registers as valid
    FSTENV  save_area          ; Save NPX environment
    FWAIT                      ; Wait for the store to complete
    XCHG    save_area + 4, bx  ; Get original tag value and set new tag value
    FLDENV  save_area          ; Force all register tags as valid. BUSY is still masked. No data
    MOV     save_area, ax      ; synchronization needed. Put original control word into NPX
    MOV     save_area + 4, bx  ; environment. Put original tag word into NPX environment
    XOR     bx, bx             ; Set initial register index
reg_store_loop:
    FSTP    saved_reg [bx]     ; Save register
    ADD     bx, type saved_reg ; Bump pointer to next register
    LOOP    reg_store_loop
                               ; All done
```

NPX_clean

```
;
; Force the NPX into a clean state with TOP matching the TOP field stored in the NPX context and all
; numeric registers tagged empty. Save_area must be the NPX environment saved earlier.
; Temp_env is a 7 word temporary area used to build a prototype NPX environment. Register ax will
; not be transparent.
;
NPX_clean:
    FINIT                      ; Put NPX into known state
    MOV     ax, save_area + 2  ; Get original status word
    AND     ax, 3800H          ; Mask out the top field
    FSTENV  temp_env           ; Format a temporary environment area with all registers
                               ; stamped empty and TOP field = 0.
    FWAIT                      ; Wait for the store to finish.
    OR      temp_env + 2, ax   ; Put in the desired TOP value.
    FLDENV  temp_env           ; Setup new NPX environment.
                               ; Now enter small_block_NPX_restore
```

small_block_NPX_restore

```
;
; Restore the NPX context without using the FRSTOR instruction. Assume the NPX context is in the
; same form as that created by an FSAVE/FNSAVE instruction, all the registers are empty, and that
; the TOP field of the NPX matches the TOP field of the NPX context. The variable save_area must
; be an NPX context save area, 47 words long. The registers bx and cx will not be transparent.
;
small_block_NPX_restore:
    MOV     cx, 8                   ; Set register count
    MOV     bx, type saved_reg*7    ; Starting offset of ST(7)
reg_load_loop:
    FLD     saved_reg [bx]          ; Get the register
    SUB     bx, type saved_reg      ; Bump pointer to next register
    LOOP    reg_load_loop
    FLDENV  save_area               ; Restore NPX context
                                    ; All done
```

APPENDIX C

OVERVIEW

Appendix C shows how floating point values can be converted to decimal ASCII character strings. The function can be called from PLM/86, PASCAL/86, FORTRAN/86, or ASM/86 functions.

Shortness, speed, and accuracy were chosen rather than providing the maximum number of significant digits possible. An attempt is made to keep integers in their own domain to avoid unnecessary conversion errors.

Using the extended precision real number format, this routine achieves a worst case accuracy of three units in the 16th decimal position for a non-integer value or integers greater than 10^{18}. This is double precision accuracy. With values having decimal exponents less than 100 in magnitude, the accuracy is one unit in the 17th decimal position.

Higher precision can be achieved with greater care in programming, larger program size, and lower performance.

Function Partitioning

Three separate modules implement the conversion. Most of the work of the conversion is done in the module FLOATING_TO_ASCII. The other modules are provided separately since they have a more general use. One of them, GET_POWER_10, is also used by the ASCII to floating point conversion routine. The other small module, TOS_STATUS, will identify what, if anything, is in the top of the numeric register stack.

Exception Considerations

Care is taken inside the function to avoid generating exceptions. Any possible numeric value will be accepted. The only exceptions possible would occur if insufficient space exists on the numeric register stack.

The value passed in the numeric stack is checked for existence, type (NAN or infinity), and status (unnormal, denormal, zero, sign). The string size is tested for a minimum and maximum value. If the top of the register stack is empty, or the string size is too small, the function will return with an error code.

Overflow and underflow is avoided inside the function for very large or very small numbers.

Special Instructions

The functions demonstrate the operation of several numeric instructions, different data types, and precision control. Shown are instructions for automatic conversion to BCD, calculating the value of 10 raised to an integer value, establishing and maintaining concurrency, data synchronization, and use of directed rounding on the NPX.

Without the extended precision data type and built-in exponential function, the double precision accuracy of this function could not be attained with the size and speed of the shown example.

The function relies on the numeric BCD data type for conversion from binary floating point to decimal. It is

not difficult to unpack the BCD digits into separate ASCII decimal digits. The major work involves scaling the floating point value to the comparatively limited range of BCD values. To print a 9-digit result requires accurately scaling the given value to an integer between 10^8 and 10^9. For example, the number +0.123456789 requires a scaling factor of 10^9 to produce the value +123456789.0 which can be stored in 9 BCD digits. The scale factor must be an exact power of 10 to avoid to changing any of the printed digit values.

These routines should exactly convert all values exactly representable in decimal in the field size given. Integer values which fit in the given string size, will not be scaled, but directly stored into the BCD form. Non-integer values exactly representable in decimal within the string size limits will also be exactly converted. For example, 0.125 is exactly representable in binary or decimal. To convert this floating point value to decimal, the scaling factor will be 1000, resulting in 125. When scaling a value, the function must keep track of where the decimal point lies in the final decimal value.

DESCRIPTION OF OPERATION

Converting a floating point number to decimal ASCII takes three major steps: identifying the magnitude of the number, scaling it for the BCD data type, and converting the BCD data type to a decimal ASCII string.

Identifying the magnitude of the result requires finding the value X such that the number is represented by $I*10^X$, where $1.0 <= I < 10.0$. Scaling the number requires multiplying it by a scaling factor 10^S, such that the result is an integer requiring no more decimal digits than provided for in the ASCII string.

Once scaled, the numeric rounding modes and BCD conversion put the number in a form easy to convert to decimal ASCII by host software.

Implementing each of these three steps requires attention to detail. To begin with, not all floating point values have a numeric meaning. Values such as infinity, indefinite, or Not A Number (NAN) may be encountered by the conversion routine. The conversion routine should recognize these values and identify them uniquely.

Special cases of numeric values also exist. Denormals, unnormals, and pseudo zero all have a numeric value but should be recognized since all of them indicate that precision was lost during some earlier calculations.

Once it has been determined that the number has a numeric value, and it is normalized setting appropriate unnormal flags, the value must be scaled to the BCD range.

Scaling the Value

To scale the number, its magnitude must be determined. It is sufficient to calculate the magnitude to an accuracy of 1 unit, or within a factor of 10 of the given value. After scaling the number, a check will be made to see if the result falls in the range expected. If not, the result can be adjusted one decimal order of magnitude up or down. The adjustment test after the scaling is necessary due to inevitable inaccuracies in the scaling value.

Since the magnitude estimate need only be close, a fast technique is used. The magnitude is estimated by multiplying the power of 2, the unbiased floating point exponent, associated with the number by $\log_{10}2$. Rounding the result to an integer will produce an estimate of sufficient accuracy. Ignoring the fraction value can introduce a maximum error of 0.32 in the result.

Using the magnitude of the value and size of the number string, the scaling factor can be calculated. Calculating the scaling factor is the most inaccurate operation of the conversion process. The relation $10^X = 2^{**}(X^*\log_2 10)$ is used for this function. The exponentiate instruction (F2XM1) will be used.

Due to restrictions on the range of values allowed by the F2XM1 instruction, the power of 2 value will be split into integer and fraction components. The relation $2^{**}(I+F) = 2^{**}I * 2^{**}F$ allows using the FSCALE instruction to recombine the 2**F value, calculated through F2XM1, and the 2**I part.

Inaccuracy in Scaling

The inaccuracy of these operations arises because of the trailing zeroes placed into the fraction value when stripping off the integer valued bits. For each integer valued bit in the power of 2 value separated from the fraction bits, one bit of precision is lost in the fraction field due to the zero fill occurring in the least significant bits.

Up to 14 bits may be lost in the fraction since the largest allowed floating point exponent value is $2^{14} - 1$.

AVOIDING UNDERFLOW AND OVERFLOW

The fraction and exponent fields of the number are separated to avoid underflow and overflow in calculating the scaling values. For example, to scale 10^{-4932} to 10^8 requires a scaling factor of 10^{4950} which cannot be represented by the NPX.

By separating the exponent and fraction, the scaling operation involves adding the exponents separate from multiplying the fractions. The exponent arithmetic will involve small integers, all easily represented by the NPX.

FINAL ADJUSTMENTS

It is possible that the power function (Get_Power_10) could produce a scaling value such that it forms a scaled result larger than the ASCII field could allow. For example, scaling 9.9999999999999999e4900 by 1.0000000000000010e-4883 would produce 1.0000000000000009e18. The scale factor is within the accuracy of the NDP and the result is within the conversion accuracy, but it cannot be represented in BCD format. This is why there is a post-scaling test on the magnitude of the result. The result can be multiplied or divided by 10, depending on whether the result was too small or too large, respectively.

Output Format

For maximum flexibility in output formats, the position of the decimal point is indicated by a binary integer called the power value. If the power value is zero, then the decimal point is assumed to be at the right of the right-most digit. Power values greater than zero indicate how many trailing zeroes are not shown. For each unit below zero, move the decimal point to the left in the string.

The last step of the conversion is storing the result in BCD and indicating where the decimal point lies. The BCD string is then unpacked into ASCII decimal characters. The ASCII sign is set corresponding to the sign of the original value.

```
LINE    SOURCE

   1    $title(Convert a floating point number to ASCII)
   2                    name    floating_to_ascii
   3                    public  floating_to_ascii
   4                    extrn   get_power_10:near,tos_status:near
   5    ;
   6    ;         This subroutine will convert the floating point number in the
   7    ;       top of the 8087 stack to an ASCII string and separate power of 10
   8    ;       scaling value (in binary).  The maximum width of the ASCII string
   9    ;       formed is controlled by a parameter which must be > 1.  Unnormal values,
  10    ;       denormal values, and psuedo zeroes will be correctly converted.
  11    ;       A returned value will indicate how many binary bits of
  12    ;       precision were lost in an unnormal or denormal value.  The magnitude
  13    ;       (in terms of binary power) of a psuedo zero will also be indicated.
  14    ;       Integers less than 10**18 in magnitude are accurately converted if the
  15    ;       destination ASCII string field is wide enough to hold all the
  16    ;       digits.  Otherwise the value is converted to scientific notation.
  17    ;
  18    ;         The status of the conversion is identified by the return value,
  19    ;       it can be:
  20    ;
  21    ;                   0       conversion complete, string_size is defined
  22    ;                   1       invalid arguments
  23    ;                   2       exact integer conversion, string_size is defined
  24    ;                   3       indefinite
  25    ;                   4       + NAN (Not A Number)
  26    ;                   5       - NAN
  27    ;                   6       + Infinity
  28    ;                   7       - Infinity
  29    ;                   8       psuedo zero found, string_size is defined
  30    ;
  31    ;         The PLM/86 calling convention is:
  32    ;
  33    ; floating_to_ascii:
  34    ;       procedure (number,denormal_ptr,string_ptr,size_ptr,field_size,
  35    ;               power_ptr) word external;
  36    ;       declare (denormal_ptr,string_ptr,power_ptr,size_ptr) pointer;
  37    ;       declare field_size word, string_size based size_ptr word;
  38    ;       declare number real;
  39    ;       declare denormal integer based denormal_ptr;
  40    ;       declare power integer based power_ptr;
  41    ;       end floating_to_ascii;
  42    ;
  43    ;         The floating point value is expected to be on the top of the NPX
  44    ;       stack.  This subroutine expects 3 free entries on the NPX stack and
  45    ;       will pop the passed value off when done.  The generated ASCII string
  46    ;       will have a leading character either '-' or '+' indicating the sign
  47    ;       of the value.  The ASCII decimal digits will immediately follow.
  48    ;       The numeric value of the ASCII string is (ASCII STRING.)*10**POWER.
```

```
49      ;       It the given number was zero, the ASCII string will contain a sign
50      ;       and a single zero chacter.  The value string_size indicates the total
51      ;       length of the ASCII string including the sign character.  String(0) will
52      ;       always hold the sign.  It is possible for string_size to be less than
53      ;       field_size.  This occurs for zeroes or integer values.  A psuedo zero
54      ;       will return a special return code.  The denormal count will indicate
55      ;       the power of two originally associated with the value.  The power of
56      ;       ten and ASCII string will be as if the value was an ordinary zero.
57      ;
58      ;         This subroutine is accurate up to a maximum of 18 decimal digits for
59      ;       integers.  Integer values will have a decimal power of zero associated
60      ;       with them.  For non integers, the result will be accurate to within 2
61      ;       decimal digits of the 16th decimal place (double precision).  The
62      ;       exponentiate instruction is also used for scaling the value into the
63      ;       range acceptable for the BCD data type.  The rounding mode in effect
64      ;       on entry to the subroutine is used for the conversion.
65      ;
66      ;         The following registers are not transparent:
67      ;
68      ;               ax bx cx dx si di flags
69      ;
70
71      ;
72      ;         Define the stack layout.
73      ;
74      bp_save         equ     word ptr [bp]
75      es_save         equ     bp_save + size bp_save
76      return_ptr      equ     es_save + size es_save
77      power_ptr       equ     return_ptr + size return_ptr
78      field_size      equ     power_ptr + size power_ptr
79      size_ptr        equ     field_size + size field_size
80      string_ptr      equ     size_ptr + size size_ptr
81      denormal_ptr    equ     string_ptr + size string_ptr
82
83      parms_size      equ     size power_ptr + size field_size + size size_ptr +
84      &                       size string_ptr + size denormal_ptr
85      ;
86      ;         Define constants used
87      ;
88      BCD_DIGITS      equ     18              ; Number of digits in bcd_value
89      WORD_SIZE       equ     2
90      BCD_SIZE        equ     10
91      MINUS           equ     1               ; Define return values
92      NAN             equ     4               ; The exact values chosen here are
93      INFINITY        equ     6               ; important.  They must correspond to
94      INDEFINITE      equ     3               ; the possible return values and be in
95      PSUEDO_ZERO     equ     8               ; the same numeric order as tested by
96      INVALID         equ     -2              ; the program.
97      ZERO            equ     -4
98      DENORMAL        equ     -6
99      UNNORMAL        equ     -8
100     NORMAL          equ     0
101     EXACT           equ     2
102     ;
103     ;         Define layout of temporary storage area.
104     ;
105     status          equ     word ptr [bp-WORD_SIZE]
106     power_two       equ     status - WORD_SIZE
107     power_ten       equ     power_two - WORD_SIZE
108     bcd_value       equ     tbyte ptr power_ten - BCD_SIZE
109     bcd_byte        equ     byte ptr bcd_value
110     fraction        equ     bcd_value
111
112     local_size      equ     size status + size power_two + size power_ten
113     &                       + size bcd_value
114     ;
115     ;         Allocate stack space for the temporaries so the stack will be big enough
116     ;
117     stack           segment stack 'stack'
118                     db      (local_size+6) dup (?)

119     stack           ends
```

```
120
121     cgroup          group   code
122     code            segment public 'code'
123                     assume  cs:cgroup
124                     extrn   power_table:qword
125     ;
126     ;       Constants used by this function.
127     ;
128                     even                    ; Optimize for 16 bits
129     const10         dw      10              ; Adjustment value for too big BCD
130     ;
131     ;       Convert the C3,C2,C1,C0 encoding from tos_status into meaningful bit
132     ;     flags and values.
133     ;
134     status_table    db      UNNORMAL, NAN, UNNORMAL + MINUS, NAN + MINUS,
135     &                       NORMAL, INFINITY, NORMAL + MINUS, INFINITY + MINUS,
136     &                       ZERO, INVALID, ZERO + MINUS, INVALID,
137     &                       DENORMAL, INVALID, DENORMAL + MINUS, INVALID
138
139     floating_to_ascii proc
140
141             call    tos_status              ; Look at status of ST(0)
142             mov     bx,ax                   ; Get descriptor from table
143             mov     al,status_table[bx]
144             cmp     al,INVALID              ; Look for empty ST(0)
145             jne     not_empty
146     ;
147     ;         ST(0) is empty!  Return the status value.
148     ;
149             ret     parms_size
150     ;
151     ;         Remove infinity from stack and exit.
152     ;
153     found_infinity:
154
155             fstp    st(0)                   ; OK to leave fstp running
156             jmp     short exit_proc
157     ;
158     ;         String space is too small!  Return invalid code.
159     ;
160     small_string:
161
162             mov     al,INVALID
163
164     exit_proc:
165
166             mov     sp,bp                   ; Free stack space
167             pop     bp                      ; Restore registers
168             pop     es
169             ret     parms_size
170     ;
171     ;         ST(0) is NAN or indefinite.  Store the value in memory and look
172     ;       at the fraction field to separate indefinite from an ordinary NAN.
173     ;
174     NAN_or_indefinite:
175
176             fstp    fraction                ; Remove value from stack for examination
177             test    al,MINUS                ; Look at sign bit
178             fwait                           ; Insure store is done
179             jz      exit_proc               ; Can't be indefinite if positive
180
```

```
181             mov     bx,0C000H               ; Match against upper 16 bits of fraction
182             sub     bx,word ptr fraction+6  ; Compare bits 63-48
183             or      bx,word ptr fraction+4  ; Bits 32-47 must be zero
184             or      bx,word ptr fraction+2  ; Bits 31-16 must be zero
185             or      bx,word ptr fraction    ; Bits 15-0 must be zero
186             jnz     exit_proc
187
188             mov     al,INDEFINITE           ; Set return value for indefinite value
189             jmp     exit_proc
190     ;
191     ;         Allocate stack space for local variables and establish parameter
192     ;       addressibility.
193     ;
194     not_empty:
195
196             push    es                      ; Save working register
197             push    bp
198             mov     bp,sp                   ; Establish stack addressibility
199             sub     sp,local_size
200
201             mov     cx,field_size           ; Check for enough string space
202             cmp     cx,2
203             jl      small_string
204
205             dec     cx                      ; Adjust for sign character
206             cmp     cx,BCD_DIGITS           ; See if string is too large for BCD
207             jbe     size_ok
208
209             mov     cx,BCD_DIGITS           ; Else set maximum string size
210
211     size_ok:
212
213             cmp     al,INFINITY             ; Look for infinity
214             jge     found_infinity          ; Return status value for + or - inf.
215
216             cmp     al,NAN                  ; Look for NAN or INDEFINITE
217             jge     NAN_or_indefinite
218     ;
219     ;         Set default return values and check that the number is normalized.
220     ;
221             fabs                            ; Use positive value only
222                                             ; sign bit in al has true sign of value
223             mov     dx,ax                   ; Save return value for later
224             xor     ax,ax                   ; Form 0 constant
225             mov     di,denormal_ptr         ; Zero denormal count
226             mov     word ptr [di],ax
227             mov     bx,power_ptr            ; Zero power of ten value
228             mov     word ptr [bx],ax
229             cmp     dl,ZERO                 ; Test for zero
230             jae     real_zero               ; Skip power code if value is zero
231
232             cmp     dl,DENORMAL             ; Look for a denormal value
233             jae     found_denormal          ; Handle it specially
234
235             fxtract                         ; Separate exponent from significand
236             cmp     dl,UNNORMAL             ; Test for unnormal value
237             jb      normal_value
238
239             sub     dl,UNNORMAL-NORMAL      ; Return normal status with correct sign
240     ;
241     ;         Normalize the fraction, adjust the power of two in ST(1) and set
242     ;       the denormal count value.
243     ;
244     ;       Assert: 0 <= ST(0) < 1.0
245     ;
246             fldl                            ; Load constant to normalize fraction
247
248     normalize_fraction:
249
250             fadd    st(1),st                ; Set integer bit in fraction
251             fsub                            ; Form normalized fraction in ST(0)
252             fxtract                         ; Power of two field will be negative
253                                             ; of denormal count
254             fxch                            ; Put denormal count in ST(0)
```

```
255             fist    word ptr [di]           ; Put negative of denormal count in memory
256             faddp   st(2),st                ; Form correct power of two in st(1)
257                                             ; OK to use word ptr [di] now
258             neg     word ptr [di]           ; Form positive denormal count
259             jnz     not_psuedo_zero
260     ;
261     ;         A psuedo zero will appear as an unnormal number.  When attempting
262     ;       to normalize it, the resultant fraction field will be zero.  Performing
263     ;       an fxtract on zero will yield a zero exponent value.
264     ;
265             fxch                            ; Put power of two value in st(0)
266             fistp   word ptr [di]           ; Set denormal count to power of two value
267                                             ; Word ptr [di] is not used by convert
268                                             ; integer, OK to leave running
269             sub     dl,NORMAL-PSUEDO_ZERO   ; Set return value saving the sign bit
270             jmp     convert_integer         ; Put zero value into memory
271     ;
272     ;         The number is a real zero, set the return value and setup for
273     ;       conversion to BCD.
274     ;
275     real_zero:
276
277             sub     dl,ZERO-NORMAL          ; Convert status to normal value
278             jmp     convert_integer         ; Treat the zero as an integer
279     ;
280     ;         The number is a denormal.  FXTRACT will not work correctly in this
281     ;       case.  To correctly separate the exponent and fraction, add a fixed
282     ;       constant to the exponent to guarantee the result is not a denormal.
283     ;
284     found_denormal:
285
286             fldl                            ; Prepare to bump exponent
287             fxch
288             fprem                           ; Force denormal to smallest representable
289                                             ; extended real format exponent
290             fxtract                         ; This will work correctly now
291     ;
292     ;         The power of the original denormal value has been safely isolated.
293     ;       Check if the fraction value is an unnormal.
294     ;
295             fxam                            ; See if the fraction is an unnormal
296             fstsw   status                  ; Save status for later
297             fxch                            ; Put exponent in ST(0)
298             fxch    st(2)                   ; Put 1.0 into ST(0), exponent in ST(2)
299             sub     dl,DENORMAL-NORMAL      ; Return normal status with correct sign
300             test    status,4400H            ; See if C3=C2=0 impling unnormal or NAN
301             jz      normalize_fraction      ; Jump if fraction is an unnormal
302
303             fstp    st(0)                   ; Remove unnecessary 1.0 from st(0)
304     ;
305     ;         Calculate the decimal magnitude associated with this number to
306     ;       within one order.  This error will always be inevitable due to
307     ;       rounding and lost precision.  As a result, we will deliberately fail
308     ;       to consider the LOG10 of the fraction value in calculating the order.
309     ;       Since the fraction will always be 1 <= F < 2, its LOG10 will not change
310     ;       the basic accuracy of the function.  To get the decimal order of magnitude,
311     ;       simply multiply the power of two by LOG10(2) and truncate the result to
312     ;       an integer.
313     ;
314     normal_value:
315     not_psuedo_zero:
316
317             fstp    fraction                ; Save the fraction field for later use
318             fist    power_two               ; Save power of two
319             fldlg2                          ; Get LOG10(2)
320                                             ; Power two is now safe to use
321             fmul                            ; Form LOG10(of exponent of number)
322             fistp   power_ten               ; Any rounding mode will work here
323     ;
324     ;         Check if the magnitude of the number rules out treating it as
325     ;       an integer.
326     ;
327     ;       CX has the maximum number of decimal digits allowed.
```

```
328     ;
329             fwait                           ; Wait for power_ten to be valid
330             mov     ax,power_ten            ; Get power of ten of value
331             sub     ax,cx                   ; Form scaling factor necessary in ax
332             ja      adjust_result           ; Jump if number will not fit
333     ;
334     ;         The number is between 1 and 10**(field_size).
335     ;       Test if it is an integer.
336     ;
337             fild    power_two               ; Restore original number
338             mov     si,dx                   ; Save return value
339             sub     dl,NORMAL-EXACT         ; Convert to exact return value
340             fld     fraction
341             fscale                          ; Form full value, this is safe here
342             fst     st(1)                   ; Copy value for compare
343             frndint                         ; Test if its an integer
344             fcomp                           ; Compare values
345             fstsw   status                  ; Save status
346             test    status,4000H            ; C3=1 implies it was an integer
347             jnz     convert_integer
348
349             fstp    st(0)                   ; Remove non integer value
350             mov     dx,si                   ; Restore original return value
351     ;
352     ;         Scale the number to within the range allowed by the BCD format.
353     ;       The scaling operation should produce a number within one decimal order
354     ;       of magnitude of the largest decimal number representable within the
355     ;       given string width.
356     ;
357     ;         The scaling power of ten value is in ax.
358     ;
359     adjust_result:
360
361             mov     word ptr [bx],ax        ; Set initial power of ten return value
362             neg     ax                      ; Subtract one for each order of
363                                             ; magnitude the value is scaled by
364             call    get_power_10            ; Scaling factor is returned as exponent
365                                             ; and fraction
366             fld     fraction                ; Get fraction
367             fmul                            ; Combine fractions
368             mov     si,cx                   ; Form power of ten of the maximum
369             shl     si,1                    ; BCD value to fit in the string
370             shl     si,1                    ; Index in si
371             shl     si,1
372             fild    power_two               ; Combine powers of two
373             faddp   st(2),st
374             fscale                          ; Form full value, exponent was safe
375             fstp    st(1)                   ; Remove exponent
376     ;
377     ;         Test the adjusted value against a table of exact powers of ten.
378     ;       The combined errors of the magnitude estimate and power function can
379     ;       result in a value one order of magnitude too small or too large to fit
380     ;       correctly in the BCD field.  To handle this problem, pretest the
381     ;       adjusted value, if it is too small or large, then adjust it by ten and
382     ;       adjust the power of ten value.
383     ;
384     test_power:
385
386             fcom    power_table[si]+type power_table; Compare against exact power
387                                             ; entry.  Use the next entry since cx
388                                             ; has been decremented by one
389             fstsw   status                  ; No wait is necessary
390             test    status,4100H            ; If C3 = C0 = 0 then too big
391             jnz     test_for_small
392
393             fidiv   const10                 ; Else adjust value
394             and     dl,not EXACT            ; Remove exact flag
395             inc     word ptr [bx]           ; Adjust power of ten value
396             jmp     short in_range          ; Convert the value to a BCD integer
397
398     test_for_small:
399
400             fcom    power_table[si]         ; Test relative size
401             fstsw   status                  ; No wait is necessary
```

```
402             test    status,100H             ; If C0 = 0 then st(0) >= lower bound
403             jz      in_range                ; Convert the value to a BCD integer
404
405             fimul   const10                 ; Adjust value into range
406             dec     word ptr [bx]           ; Adjust power of ten value
407
408     in_range:
409
410             frndint                         ; Form integer value
411     ;
412     ;       Assert: 0 <= TOS <= 999,999,999,999,999,999
413     ;       The TOS number will be exactly representable in 18 digit BCD format.
414     ;
415     convert_integer:
416
417             fbstp   bcd_value               ; Store as BCD format number
418     ;
419     ;         While the store BCD runs, setup registers for the conversion to
420     ;       ASCII.
421     ;
422             mov     si,BCD_SIZE-2           ; Initial BCD index value
423             mov     cx,0f04h                ; Set shift count and mask
424             mov     bx,1                    ; Set initial size of ASCII field for sign
425             mov     di,string_ptr           ; Get address of start of ASCII string
426             mov     ax,ds                   ; Copy ds to es
427             mov     es,ax
428             cld                             ; Set autoincrement mode
429             mov     al,'+'                  ; Clear sign field
430             test    dl,MINUS                ; Look for negative value
431             jz      positive_result
432
433             mov     al,'-'
434
435     positive_result:
436
437             stosb                           ; Bump string pointer past sign
438             and     dl,not MINUS            ; Turn off sign bit
439             fwait                           ; Wait for fbstp to finish
440     ;
441     ;         Register usage:
442     ;                                 ah:     BCD byte value in use
443     ;                                 al:     ASCII character value
444     ;                                 dx:     Return value
445     ;                                 ch:     BCD mask = 0fh
446     ;                                 cl:     BCD shift count = 4
447     ;                                 bx:     ASCII string field width
448     ;                                 si:     BCD field index
449     ;                                 di:     ASCII string field pointer
450     ;                                 ds,es:  ASCII string segment base
451     ;
452     ;         Remove leading zeroes from the number.
453     ;
454     skip_leading_zeroes:
455
456             mov     ah,bcd_byte[si]         ; Get BCD byte
457             mov     al,ah                   ; Copy value
458             shr     al,cl                   ; Get high order digit
459             and     al,ch                   ; Set zero flag
460             jnz     enter_odd               ; Exit loop if leading non zero found
461
462             mov     al,ah                   ; Get BCD byte again
463             and     al,ch                   ; Get low order digit
464             jnz     enter_even              ; Exit loop if non zero digit found
465
466             dec     si                      ; Decrement BCD index
467             jns     skip_leading_zeroes
468     ;
469     ;         The significand was all zeroes.
470     ;
471             mov     al,'0'                  ; Set initial zero
472             stosb
473             inc     bx                      ; Bump string length
474             jmp     short exit_with_value
```

```
475     ;
476     ;         Now expand the BCD string into digit per byte values 0-9.
477     ;
478     digit_loop:
479
480             mov     ah,bcd_byte[si]         ; Get BCD byte
481             mov     al,ah
482             shr     al,cl                   ; Get high order digit
483
484     enter_odd:
485
486             add     al,'0'                  ; Convert to ASCII
487             stosb                           ; Put digit into ASCII string area
488             mov     al,ah                   ; Get low order digit
489             and     al,ch
490             inc     bx                      ; Bump field size counter
491
492     enter_even:
493
494             add     al,'0'                  ; Convert to ASCII
495             stosb                           ; Put digit into ASCII area
496             inc     bx                      ; Bump field size counter
497             dec     si                      ; Go to next BCD byte
498             jns     digit_loop
499     ;
500     ;         Conversion complete.  Set the string size and remainder.
501     ;
502     exit_with_value:
503
504             mov     di,size_ptr
505             mov     word ptr [di],bx
506             mov     ax,dx                   ; Set return value
507             jmp     exit_proc
508
509     floating_to_ascii       endp
510     code                    ends
511                             end

ASSEMBLY COMPLETE, NO ERRORS FOUND


LINE    SOURCE

   1    $title(Calculate the value of 10**ax)
   2    ;
   3    ;         This subroutine will calculate the value of 10**ax.
   4    ;       All 8086 registers are transparent and the value is returned on
   5    ;       the TOS as two numbers, exponent in ST(1) and fraction in ST(0).
   6    ;       The exponent value can be larger than the maximum representable
   7    ;       exponent.  Three stack entries are used.
   8    ;
   9                    name    get_power_10
  10                    public  get_power_10,power_table
  11
  12    stack           segment stack  'stack'
  13                    dw      4 dup (?)               ; Allocate space on the stack

  14    stack           ends
  15
  16    cgroup          group   code
  17    code            segment public 'code'
  18                    assume  cs:cgroup
  19    ;
  20    ;         Use exact values from 1.0 to 1e18.
  21    ;
  22                    even                            ; Optimize 16 bit access
  23    power_table     dq      1.0,1e1,1e2,1e3
```

```
24                      dq      1e4,1e5,1e6,1e7

25                      dq      1e8,1e9,1e10,1e11

26                      dq      1e12,1e13,1e14,1e15

27                      dq      1e16,1e17,1e18

28
29      get_power_10    proc
30
31              cmp     ax,18                   ; Test for 0 <= ax < 19
32              ja      out_of_range
33
34              push    bx                      ; Get working index register
35              mov     bx,ax                   ; Form table index
36              shl     bx,1
37              shl     bx,1
38              shl     bx,1
39              fld     power_table[bx]         ; Get exact value
40              pop     bx                      ; Restore register value
41              fxtract                         ; Separate power and fraction
42              ret                             ; OK to leave fxtract running
43      ;
44      ;         Calculate the value using the exponentiate instruction.
45      ;       The following relations are used:
46      ;               10**x = 2**(log2(10)*x)
47      ;               2**(I+F) = 2**I * 2**F
48      ;               if st(1) = I and st(0) = 2**F then fscale produces 2**(I+F)
49      ;
50      out_of_range:
51
52              fldl2t                          ; TOS = LOG2(10)
53              push    bp                      ; Establish stack addressibility
54              mov     bp,sp
55              push    ax                      ; Put power (P) in memory
56              push    ax                      ; Allocate space for status
57              fimul   word ptr [bp-2]         ; TOS,X = LOG2(10)*P = LOG2(10**P)
58              fnstcw  word ptr [bp-4]         ; Get current control word
59                                              ; Control word is a static value
60              mov     ax,word ptr [bp-4]      ; Get control word, no wait necessary
61              and     ax,not 0C00H            ; Mask off current rounding field
62              or      ax,0400H                ; Set round to negative infinity
63              xchg    ax,word ptr [bp-4]      ; Put new control word in memory
64                                              ; old control word is in ax
65              fld1                            ; Set TOS = -1.0
66              fchs
67              fld     st(1)                   ; Copy power value in base two
68              fldcw   word ptr [bp-4]         ; Set new control word value
69              frndint                         ; TOS = I: -inf < I <= X, I is an integer
70              mov     word ptr [bp-4],ax      ; Restore original rounding control
71              fldcw   word ptr [bp-4]
```

```
 72              fxch    st(2)                   ; TOS = X, ST(1) = -1.0, ST(2) = I
 73              pop     ax                      ; Remove original control word
 74              fsub    st,st(2)                ; TOS,F = X-I: 0 <= TOS < 1.0
 75              pop     ax                      ; Restore power of ten
 76              fscale                          ; TOS = F/2: 0 <= TOS < 0.5
 77              f2xm1                           ; TOS = 2**(F/2) - 1.0
 78              pop     bp                      ; Restore stack
 79              fsubr                           ; Form 2**(F/2)
 80              fmul    st,st(0)                ; Form 2**F
 81              ret                             ; OK to leave fmul running
 82
 83      get_power_10    endp
 84      code            ends
 85                      end

ASSEMBLY COMPLETE, NO ERRORS FOUND


LINE    SOURCE

  1     $title(Determine TOS register contents)
  2     ;
  3     ;         This subroutine will return a value from 0-15 in ax corresponding
  4     ;       to the contents of 8087 TOS.  All registers are transparent and no
  5     ;       errors are possible.  The return value corresponds to c3,c2,c1,c0
  6     ;       of FXAM instruction.
  7     ;
  8             name    tos_status
  9             public  tos_status
 10
 11     stack            segment stack 'stack'
 12                      dw      3 dup (?)       ; Allocate space on the stack


 13     stack            ends
 14
 15     cgroup           group   code
 16     code             segment public 'code'
 17                      assume  cs:cgroup
 18     tos_status       proc
 19
 20             fxam                    ; Get register contents status
 21             push    ax              ; Allocate space for status value
 22             push    bp              ; Establish stack addressibility
 23             mov     bp,sp
 24             fstsw   word ptr [bp+2] ; Put tos status in memory
 25             pop     bp              ; Restore registers
 26             pop     ax              ; Get status value, no wait necessary
 27             mov     al,ah           ; Put bit 10-8 into bits 2-0
 28             and     ax,4007h        ; Mask out bits c3,c2,c1,c0
 29             shr     ah,1            ; Put bit c3 into bit 11
 30             shr     ah,1
 31             shr     ah,1
 32             or      al,ah           ; Put c3 into bit 3
 33             mov     ah,0            ; Clear return value
 34             ret
 35
 36     tos_status       endp
 37     code             ends
 38                      end

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPENDIX D

OVERVIEW

Appendix D shows a function for converting ASCII input strings into floating point values. The returned value can be used by PLM/86, PASCAL/86, FORTRAN/86, or ASM/86. The routine will accept a number in ASCII of standard FORTRAN formats. Up to 18 decimal digits are accepted and the conversion accuracy is the same as for converting in the other direction. Greater accuracy can also be achieved with similar tradeoffs, as mentioned earlier.

Description of Operation

Converting from ASCII to floating point is less complex numerically than going from floating point to ASCII. It consists of four basic steps: determine the size in decimal digits of the number, build a BCD value corresponding to the number string if the decimal point were at the far right, calculate the exponent value, and scale the BCD value. The first three steps are performed by the host software. The fourth step is mainly performed by numeric operations.

The complexity in this function arises due to the flexible nature of the input values it will recognize. Most of the code simply determines the meaning of each character encountered. Two separate number inputs must be recognized, mantissa and exponent values. Performing the numerics operations is very straightforward.

The length of the number string is determined first to allow building a BCD number from low digits to high digits. This technique guarantees that an integer will be converted to its exact BCD integer equivalent.

If the number is a floating point value, then the digit string can be scaled appropriately. If a decimal point occurs within the string, the scale factor must be decreased by one for each digit the decimal point is moved to the right. This factor must be added to any exponent value specified in the number.

ACCURACY CONSIDERATIONS

All the same considerations for converting floating point to ASCII apply to calculating the scaling factor. The accuracy of the scale factor determines the accuracy of the result.

The exponents and fractions are again kept separate to prevent overflows or underflows during the scaling operations.

```
LINE    SOURCE

   1    $title(ASCII to floating point conversion)
   2    ;
   3    ;         Define the publicly known names.
   4    ;
   5                  name    ascii_to_floating
   6                  public  ascii_to_floating
   7                  extrn   get_power_10:near
   8    ;
   9    ;         This function will convert an ASCII character string to a floating
  10    ;       point representation.  Character strings in integer or scientific form
  11    ;       will be accepted.  The allowed format is:
  12    ;
  13    ;       [+,-][digit(s)][.][digit(s)][E,e][+,-][digit(s)]
  14    ;
  15    ;          Where a digit must have been encountered before the exponent
  16    ;       indicator 'E' or'e'.  If a '+', '-', or '.' was encountered, then at
  17    ;       least one digit must exist before the optional exponent field.  A value
  18    ;       will always be returned in the 8087 stack.  In case of invalid numbers,
  19    ;       values like indefinite or infinity will be returned.
  20    ;
  21    ;         The first character not fitting within the format will terminate the
  22    ;       conversion.  The address of the terminating character will be returned
  23    ;       by this subroutine.
  24    ;
  25    ;         The result will be left on the top of the NPX stack.  This
  26    ;       subroutine expects 3 free NPX stack registers.  The sign of the result
  27    ;       will correspond to any sign characters in the ASCII string.  The rounding
  28    ;       mode in effect at the time the subroutine was called will be used for
  29    ;       the conversion from base 10 to base 2.  Up to 18 significant decimal
  30    ;       digits may appear in the number.  Leading zeroes, trailing zeroes, or
  31    ;       exponent digits do not count towards the 18 digit maximum.  Integers
  32    ;       or exactly representable decimal numbers of 18 digits or less will be
  33    ;       exactly converted.  The technique used constructs a BCD number
```

```
34      ;       representing the significant ASCII digits of the string with the decimal
35      ;       point removed.
36      ;
37      ;         An attempt is made to exactly convert relatively small integers or
38      ;       small fractions.  For example the values: .06125, 123456789012345678,
39      ;       1e17, 1.23456e5, and 125e-3 will be exactly converted to floating point.
40      ;         The exponentiate instruction is used to scale the generated BCD vaslue
41      ;       to very large or very small numbers.  The basic accuracy of this function
42      ;       determines the accuracy of this subroutine.  For very large or very small
43      ;       numbers, the accuracy of this function is 2 units in the 16th decimal
44      ;       place or double precision.  The range of decimal powers accepted is
45      ;       10**-4930 to 10**4930.
46      ;
47      ;         The PLM/86 calling format is:
48      ;
49      ;       ascii_to_floating:
50      ;               procedure (string_ptr,end_ptr,status_ptr) real external;
51      ;               declare (string_ptr,end_ptr,status_ptr) pointer;
52      ;               declare end based end_ptr pointer;
53      ;               declare status based status_ptr word;
54      ;               end;
55      ;
56      ;         The status value has 6 possible states:
57      ;
58      ;               0       A number was found.
59      ;               1       No number was found, return indefinite.
60      ;               2       Exponent was expected but none found, return indefinite.
61      ;               3       Too many digits were found, return indefinite.
62      ;               4       Exponent was too big, return a signed infinity.
63      ;
64      ;         The following registers are used by this subroutine:
65      ;
66      ;       ax      bx      cx      dx      si      di
67      ;
68
69      ;
70      ;         Define constants.
71      ;
72      LOW_EXPONENT    equ     -4930                   ; Smallest allowed power of 10
73      HIGH_EXPONENT   equ     4930                    ; Largest allowed power of 10
74      WORD_SIZE       equ     2
75      BCD_SIZE        equ     10
76      ;
77      ;         Define the parameter layouts involved:
78      ;
79      bp_save         equ     word ptr [bp]
80      return_ptr      equ     bp_save + size bp_save
81      status_ptr      equ     return_ptr + size return_ptr
82      end_ptr         equ     status_ptr + size status_ptr
83      string_ptr      equ     end_ptr + size end_ptr
84
85      parms_size      equ     size status_ptr + size end_ptr + size string_ptr
86      ;
87      ;         Define the local variable data layouts
88      ;
89      power_ten       equ     word ptr [bp- WORD_SIZE] ; power of ten value
90      bcd_form        equ     tbyte ptr power_ten - BCD_SIZE; BCD representation
91
92      local_size      equ     size power_ten + size bcd_form
93      ;
94      ;         Define common expressions used
95      ;
96      bcd_byte        equ     byte ptr bcd_form       ; Current byte in the BCD form
97      bcd_count       equ     (type(bcd_form)-1)*2    ; Number of digits in BCD form
98      bcd_sign        equ     byte ptr bcd_form + 9   ; Address of BCD sign byte
99      bcd_sign_bit    equ     80H
100     ;
101     ;         Define return values.
102     ;
103     NUMBER_FOUND    equ     0               ; Number was found
104     NO_NUMBER       equ     1               ; No number was found
105     NO_EXPONENT     equ     2               ; No exponent was found when expected
106     TOO_MANY_DIGITS equ     3               ; Too many digits were found
107     EXPONENT_TOO_BIG equ    4               ; Exponent was too big
```

```
108     ;
109     ;         Allocate stack space to insure enough exists at run time.
110     ;
111     stack           segment stack 'stack'
112                     db      (local_size+4) dup (?)

113     stack           ends
114
115     cgroup          group   code
116     code            segment public 'code'
117                     assume  cs:cgroup
118     ;
119     ;         Define some of the possible return values.
120     ;
121                     even                            ; Optimize 16 bit access
122     indefinite      dd      0FFC00000R              ; Single precision real for indefinite
123     infinity        dd      07FF80000R              ; Single precision real for +infinity
124
125     ascii_to_floating proc
126
127             fldz                                    ; Prepare to zero BCD value
128             push    bp                              ; Save callers stack environment
129             mov     bp,sp                           ; Establish stack addressibility
130             sub     sp,local_size                   ; Allocate space for local variables
131     ;
132     ;         Get any leading sign character to form initial BCD template.
133     ;
134             mov     si,string_ptr                   ; Get starting address of the number
135             xor     dx,dx                           ; Set initial decimal digit count
136             cld                                     ; Set autoincrement mode
137     ;
138     ;         Register usage:
139     ;
140     ;               al:     Current character value being examined
141     ;               cx:     Digit count before the decimal point
142     ;               dx:     Total digit count
143     ;               si:     Pointer to character string
144     ;
145     ;         Look for an initial sign and skip it if found.
146     ;
147             lodsb                                   ; Get first character
148             cmp     al,'+'                          ; Look for a sign
149             jz      scan_leading_digits
150
151             cmp     al,'-'
152             jnz     enter_leading_digits            ; If not "-" test current character
153
154             fchs                                    ; Set TOS = -0
155     ;
156     ;         Count the number of digits appearing before an optional decimal point.
157     ;
158     scan_leading_digits:
159
160             lodsb                                   ; Get next character
161
162     enter_leading_digits:
163
164             call    test_digit                      ; Test for digit and bump counter
165             jnc     scan_leading_digits
166     ;
167     ;         Look for a possible decimal point and start fbstp operation.
168     ;       The fbstp zeroes out the BCD value and sets the correct sign.
169     ;
170             fbstp   bcd_form                        ; Set initial sign and value of BCD number
171             mov     cx,dx                           ; Save count of digits before decimal point
172             cmp     al,'.'
173             jnz     test_for_digits
174     ;
175     ;         Count the number of digits appearing after the decimal point.
176     ;
177     scan_trailing_digits:
178
179             lodsb                                   ; Look at next character
```

```
180             call    test_digit              ; Test for digit and bump counter
181             jnc     scan_trailing_digits
182     ;
183     ;         There must be at least one digit counted at this point.
184     ;
185     test_for_digits:
186
187             dec     si                      ; Put si back on terminating character
188             or      dx,dx                   ; Test digit count
189             jz      no_number_found         ; Jump if no digits were found
190
191             push    si                      ; Save pointer to terminator
192             dec     si                      ; Backup pointer to last digit
193     ;
194     ;         Check that the number will fit in the 18 digit BCD format.
195     ;       CX becomes the initial scaling factor to account for the implied
196     ;       decimal point.
197     ;
198             sub     cx,dx                   ; For each digit to the right of the
199                                             ; decimal point, subtract one from the
200                                             ; initial scaling power
201             neg     dx                      ; Use negative digit count so the
202                                             ; test_digit routine can count dx up
203                                             ; to zero
204             cmp     dx,-bcd_count           ; See if too many digits found
205             jb      test_for_unneeded_digits
206     ;
207     ;         Setup initial register values for scanning the number right to left
208     ;       while building the BCD value in memory.
209     ;
210     form_bcd_value:
211
212             std                             ; Set autodecrement mode
213             mov     power_ten,cx            ; Set initial power of ten
214             xor     di,di                   ; Clear BCD number index
215             mov     cl,4                    ; Set digit shift count
216             fwait                           ; Ensure BCD store is done
217             jmp     enter_digit_loop
218     ;
219     ;         No digits were encountered before testing for the exponent.
220     ;       Restore the string pointer and return an indefinite value.
221     ;
222     no_number_found:
223
224             mov     ax,NO_NUMBER            ; Set return status
225             fld     indefinite              ; Return an indefinite numeric value
226             jmp     exit
227     ;
228     ;         Test for a number of the form ???00000.
229     ;
230     test_terminating_point:
231
232             lodsb                           ; Get last character
233             cmp     al,'.'                  ; Look for decimal point
234             jz      enter_power_zeroes      ; Skip forward if found
235
236             inc     si                      ; Else bump pointer back
237             jmp     short enter_power_zeroes
238     ;
239     ;         Too many decimal digits encountered.  Attempt to remove leading and
240     ;       trailing digits to bring the total into the bounds of the BCD format.
241     ;
242     test_for_unneeded_digits:
243
244             std                             ; Set autodecrement mode
245             or      cx,cx                   ; See if any digits appeared to the
246                                             ; right of the decimal point
247             jz      test_terminating_point  ; Jump if none exist
248
249             dec     dx                      ; Adjust digit counter for loop
250     ;
251     ;         Scan backwards from the right skipping trailing zeroes.
252     ;       If the end of the number is encountered, dx=0, the string consists of
253     ;       all zeroes!
```

```
254     ;
255     skip_trailing_zeroes:
256
257             inc     dx                      ; Bump digit count
258             jz      look_for_exponent       ; Jump if string of zeroes found!
259
260             lodsb                           ; Get next character
261             inc     cx                      ; Bump power value for each trailing
262             cmp     al,'0'                  ; zero dropped
263             jz      skip_trailing_zeroes
264
265             dec     cx                      ; Adjust power counter from loop
266             cmp     al,'.'                  ; Look for decimal point
267             jnz     scan_leading_zeroes     ; Skip forward if none found
268
269             dec     dx                      ; Adjust counter for the decimal point
270     ;
271     ;         The string is of the form: ????.0000000
272     ;       See if any zeroes exist to the left of the decimal point.
273     ;
274     enter_power_zeroes:
275
276             dec     dx                      ; Adjust digit counter for loop
277
278     skip_power_zeroes:
279
280             inc     dx                      ; Bump digit count
281             jz      look_for_exponent
282
283             lodsb                           ; Get next character
284             inc     cx                      ; Bump power value for each trailing
285             cmp     al,'0'                  ; zero dropped
286             jz      skip_power_zeroes
287
288             dec     cx                      ; Adjust power counter from loop
289     ;
290     ;         Scan the leading digits from the left to see if they are zeroes.
291     ;
292     scan_leading_zeroes:
293
294             lea     di,byte ptr [si+1]      ; Save new end of number pointer
295             cld                             ; Set autoincrement mode
296             mov     si,string_ptr           ; Set pointer to the start
297             lodsb                           ; Look for sign character
298             cmp     al,'+'
299             je      skip_leading_zeroes
300
301             cmp     al,'-'
302             jne     enter_leading_zeroes
303     ;
304     ;         Drop leading zeroes.  None of them affect the power value in cx.
305     ;       We are guarenteed at least one non zero digit to terminate the loop.
306     ;
307     skip_leading_zeroes:
308
309             lodsb                           ; Get next character
310
311     enter_leading_zeroes:
312
313             inc     dx                      ; Bump digit count
314             cmp     al,'0'                  ; Look for a zero
315             jz      skip_leading_zeroes
316
317             dec     dx                      ; Adjust digit count from loop
318             cmp     al,'.'                  ; Look for 000.??? form
319             jnz     test_digit_count
320     ;
321     ;         Number is of the form 000.????
322     ;       Drop all leading zeroes with no effect on the power value.
323     ;
324     skip_middle_zeroes:
325
326             inc     dx                      ; Remove the digit
327             lodsb                           ; Get next character
```

```
328             cmp     al,'0'
329             jz      skip_middle_zeroes
330
331             dec     dx                      ; Adjust digit count from loop
332     ;
333     ;         All superflous zeroes are removed.  Check if all is well now.
334     ;
335     test_digit_count:
336
337             cmp     dx,-bcd_count
338             jb      too_many_digits_found
339
340             mov     si,di                   ; Restore string pointer
341             jmp     form_bcd_value
342
343     too_many_digits_found:
344
345             fld     indefinite              ; Set return numeric value
346             mov     ax,TOO_MANY_DIGITS      ; Set return flag
347             pop     si                      ; Get last address
348             jmp     exit
349     ;
350     ;         Build BCD form of the decimal ASCII string from right to left with
351     ;       trailing zeroes and decimal point removed.  Note that the only non
352     ;       digit possible is a decimal point which can be safely ignored.
353     ;       Test_digit will correctly count dx back towards zero to terminate
354     ;       the BCD build function.
355     ;
356     get_digit_loop:
357
358             lodsb                           ; Get next character
359             call    test_digit              ; Check if digit and bump digit count
360             jc      get_digit_loop          ; Skip the decimal point if found
361
362             shl     al,cl                   ; Put digit into high nibble
363             or      ah,al                   ; Form BCD byte in ah
364             mov     bcd_byte[di],ah         ; Put into BCD string
365             inc     di                      ; Bump BCD pointer
366             or      dx,dx                   ; Check if digit is available
367             jz      look_for_exponent
368
369     enter_digit_loop:
370
371             lodsb                           ; Get next character
372             call    test_digit              ; Check if digit
373             jc      enter_digit_loop        ; Skip the decimal point
374
375             mov     ah,al                   ; Save digit
376             or      dx,dx                   ; Check if digit is available
377             jnz     get_digit_loop
378
379             mov     bcd_byte[di],ah         ; Save last odd digit
380     ;
381     ;         Look for an exponent indicator.
382     ;
383     look_for_exponent:
384
385             pop     si                      ; Restore string pointer
386             cld                             ; Set autoincrement direction
387             mov     di,power_ten            ; Get current power of ten
388             lodsb                           ; Get next character
389             cmp     al,'e'                  ; Look for exponent indication
390             je      exponent_found
391
392             cmp     al,'E'
393             jne     convert
394     ;
395     ;         An exponent is expected, get its numeric value.
396     ;
397     exponent_found:
398
399             lodsb                           ; Get next character
400             xor     di,di                   ; Clear power variable
401             mov     cx,di                   ; Clear exponent sign flag and digit flag
```

```
402             cmp     al,'+'                  ; Test for positive sign
403             je      skip_power_sign
404
405             cmp     al,'-'                  ; Test for negative sign
406             jne     enter_power_loop
407     ;
408     ;         The exponent is negative.
409     ;
410             inc     ch                      ; Set exponent sign flag
411
412     skip_power_sign:
413     ;
414     ;         Register usage:
415     ;
416     ;               al:     exponent character being examined
417     ;               bx:     return value
418     ;               ch:     exponent sign flag      0 positive, 1 negative
419     ;               cl:     digit flag      0 no digits found, 1 digits found
420     ;               dx:     not usable since test_digit increments it
421     ;               si:     string pointer
422     ;               di:     binary value of exponent
423     ;
424     ;         Scan off exponent digits until a non-digit is encountered.
425     ;
426     power_loop:
427
428             lodsb                           ; Get next character
429
430     enter_power_loop:
431
432             mov     ah,0                    ; Clear ah since ax is added to later
433             call    test_digit              ; Test for a digit
434             jc      form_power_value        ; Exit loop if not
435
436             mov     cl,1                    ; Set power digit flag
437             sal     di,1                    ; old*2
438             add     ax,di                   ; old*2+digit
439             sal     di,1                    ; old*4
440             sal     di,1                    ; old*8
441             add     di,ax                   ; old*10+digit
442             cmp     di,HIGH_EXPONENT+bcd_count; Check if exponent is too big
443             jna     power_loop
444     ;
445     ;         The exponent is too large.
446     ;
447     exponent_overflow:
448
449             mov     ax,EXPONENT_TOO_BIG     ; Set return value
450             fld     infinity                ; Return infinity
451             test    bcd_sign,bcd_sign_bit   ; Return correctly signed infinity
452             jz      exit                    ; Jump if not
453
454             fchs                            ; Return -infinity
455             jmp     short exit
456     ;
457     ;         No exponent was found.
458     ;
459     no_exponent_found:
460
461             dec     si                      ; Put si back on terminating character
462             mov     ax,NO_EXPONENT          ; Set return value
463             fld     indefinite              ; Set number to return
464             jmp     short exit
465     ;
466     ;         The string examination is complete.  Form the correct power of ten.
467     ;
468     form_power_value:
469
470             dec     si                      ; Backup string pointer to terminating
471                                             ; character
472             rcr     ch,1                    ; Test exponent sign flag
473             jnc     positive_exponent
474
475             neg     di                      ; Force exponent negative
```

```
476
477     positive_exponent:
478
479             rcr     cl,1                    ; Test exponent digit flag
480             jnc     no_exponent_found       ; If zero then no exponent digits were
481                                             ; found
482             add     di,power_ten            ; Form the final power of ten value
483             cmp     di,LOW_EXPONENT         ; Check if the value is in range
484             js      exponent_overflow       ; Jump if exponent is too small
485
486             cmp     di,HIGH_EXPONENT
487             jg      exponent_overflow
488
489             inc     si                      ; Adjust string pointer
490     ;
491     ;         Convert the base 10 number to base 2.
492     ;       Note:   10**exp = 2**(exp*log2(10))
493     ;
494     ;       di has binary power of ten value to scale the BCD value with.
495     ;
496     convert:
497
498             dec     si                      ; Bump string pointer back to last character
499             mov     ax,di                   ; Set power of ten to calculate
500             or      ax,ax                   ; Test for positive or negative value
501             js      get_negative_power
502     ;
503     ;         Scale the BCD value by a value >= 1.
504     ;
505             call    get_power_10            ; Get the adjustment power of ten
506             fbld    bcd_form                ; Get the digits to use
507             fmul                            ; Form converged result
508             jmp     short done
509     ;
510     ;         Calculate a power of ten value > 1 then divide the BCD value with
511     ;       it.  This technique is more exact than multiplying the BCD value by
512     ;       a fraction since no negative power of ten can be exactly represented
513     ;       in binary floating point.  Using this technique will guarentee exact
514     ;       conversion of values like .5 and .0625.
515     ;
516     get_negative_power:
517
518             neg     ax                      ; Force positive power
519             call    get_power_10            ; Get the adjustment power of ten
520             fbld    bcd_form                ; Get the digits to use
521             fdivr                           ; Divide fractions
522             fxch                            ; Negate scale factor
523             fchs
524             fxch
525     ;
526     ;         All done, set return values.
527     ;
528     done:
529
530             fscale                          ; Update exponent of the result
531             mov     ax,NUMBER_FOUND         ; Set return value
532             fstp    st(1)                   ; Remove the scale factor
533
534     exit:
535
536             mov     di,status_ptr           ; Set status of the conversion
537             mov     word ptr [di],ax
538             mov     di,end_ptr              ; Set ending string address
539             mov     word ptr [di],si
540             mov     sp,bp                   ; Deallocate local storage area
541             pop     bp                      ; Restore caller's environment
542             fwait                           ; Insure all loads from memory are done
543             ret     parms_size
544     ;
545     ;         Test if the character in al is an ASCII digit.
546     ;       If so then convert to binary, bump cx, and clear the carry flag.
547     ;       Else leave as is and set the carry flag.
```

```
548     ;
549     test_digit:
550             cmp     al,'9'                  ; See if a digit
551             ja      not_digit
552
553             cmp     al,'0'
554             jb      not_digit
555     ;
556     ;         Character is a digit.
557     ;
558             inc     dx                      ; Bump digit count
559             sub     al,'0'                  ; Convert to binary and clear carry flag
560             ret
561     ;
562     ;         Character is not a digit
563     ;
564     not_digit:
565             stc                             ; Leave as is and set the carry flag
566             ret
567
568     ascii_to_floating endp
569     code            ends
570                     end

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPENDIX E

OVERVIEW

Appendix E contains three trigonometric functions for sine, cosine, and tangent. All accept a valid angle argument between -2^{62} and $+2^{62}$. They may be called from PLM/86, PASCAL/86, FORTRAN/86 or ASM/86 functions.

They use the partial tangent instruction together with trigonometric identities to calculate the result. They are accurate to within 16 units of the low 4 bits of an extended precision value. The functions are coded for speed and small size, with tradeoffs available for greater accuracy.

FPTAN and FPREM

These trigonometric functions use the FPTAN instruction of the NPX. FPTAN requires that the angle argument be between 0 and PI/4 radians, 0 to 45 degrees. The FPREM instruction is used to reduce the argument down to this range. The low three quotient bits set by FPREM identify which octant the original angle was in.

One FPREM instruction iteration can reduce angles of 10^{18} radians or less in magnitude to PI/4! Larger values can be reduced, but the meaning of the result is questionable since any errors in the least significant bits of that value represent changes of 45 degrees or more in the reduced angle.

Cosine Uses Sine Code

To save code space, the cosine function uses most of the sine function code. The relation sin (|A| + PI/2) = cos(A) is used to convert the cosine argument into a sine argument. Adding PI/2 to the angle is performed by adding 010_2 to the FPREM quotient bits identifying the argument's octant.

It would be very inaccurate to add PI/2 to the cosine argument if it was very much different from PI/2.

Depending on which octant the argument falls in, a different relation will be used in the sine and tangent functions. The program listings show which relations are used.

For the tangent function, the ratio produced by FPTAN will be directly evaluated. The sine function will use either a sine or cosine relation depending on which octant the angle fell into. On exit these functions will normally leave a divide instruction in progress to maintain concurrency.

If the input angles are of a restricted range, such as from 0 to 45 degrees, then considerable optimization is possible since full angle reduction and octant identification is not necessary.

All three functions begin by looking at the value given to them. Not a number (NAN), infinity, or empty registers must be specially treated. Unnormals need to be converted to normal values before the FPTAN instruction will work correctly. Denormals will be converted to very small unnormals which do work correctly for the FPTAN instruction. The sign of the angle is saved to control the sign of the result.

Within the functions, close attention was paid to maintain concurrent execution of the 8087 and host. The concurrent execution will effectively hide the execution time of the decision logic used in the program.

```
LINE     SOURCE

   1     $title(8087 Trignometric Functions)
   2
   3                     public          sine,cosine,tangent
   4                     name            trig_functions
   5
   6 +1  $include (:fl:8087.anc)
   7     ;
   8     ;          Define 8087 word packing in the environment area.
   9     ;
  10     cw_87              record  res871:3,infinity_control:1,rounding_control:2,
  11     &                          precision_control:2,error_enable:1,res872:1,
  12     &                          precision_mask:1,underflow_mask:1,overflow_mask:1,
  13     &                          zero_divide_mask:1,denormal_mask:1,invalid_mask:1
  14
  15     sw_87              record  busy:1,cond3:1,top:3,cond2:1,cond1:1,cond0:1,
  16     &                          error_pending:1,res873:1,precision_error:1,
  17     &                          underflow_error:1,overflow_error:1,zero_divide_error:1,
  18     &                          denormal_error:1,invalid_error:1
  19
  20     tw_87              record  reg7_tag:2,reg6_tag:2,reg5_tag:2,reg4_tag:2,
  21     &                          reg3_tag:2,reg2_tag:2,reg1_tag:2,reg0_tag:2
  22
  23     low_ip_87          record  low_ip:16
  24
  25     high_ip_op_87      record  hi_ip:4,res874:1,opcode_87:11
  26
  27     low_op_87          record  low_op:16
  28
  29     high_op_87         record  hi_op:4,res875:12
  30
  31     environment_87     struc                   ; 8087 environemnt layout
  32     env87_cw           dw              ?
  33     env87_sw           dw              ?
  34     env87_tw           dw              ?
  35     env87_low_ip       dw              ?
  36     env87_hip_op       dw              ?
  37     env87_low_op       dw              ?
  38     env87_hop          dw              ?
  39     environment_87     ends
  40     ;
  41     ;          Define 8087 related constants.
  42     ;
  43     TOP_VALUE_INC      equ             sw_87 <0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0>
  44
  45     VALID_TAG          equ             0       ; Tag register values
  46     ZERO_TAG           equ             1
  47     SPECIAL_TAG        equ             2
  48     EMPTY_TAG          equ             3
  49     REGISTER_MASK      equ             7
  50
  51     ;
  52     ;          Define local variable areas.
  53     ;
  54     stack              segment stack 'stack'
  55
  56     local_area         struc
  57     sw1                dw      ?               ; 8087 status value
  58     local_area         ends
  59
  60                        db      size local_area+4       ; Allocate stack space
  61     stack              ends
  62
  63     code               segment public 'code'
  64                        assume  cs:code,ss:stack
  65     ;
  66     ;          Define local constants.
  67     ;
  68     status             equ     [bp].sw1                ; 8087 status value location
  69
  70                        even
  71
  72     pi_quarter         dt      3FFEC90FDAA22168C235R   ; PI/4
```

```
indefinite      dd      0FFC00000R              ; Indefinite special value

;
;         This subroutine calculates the sine or cosine of the angle, given in
;       radians.  The angle is in ST(0), the returned value will be in ST(0).
;       The result is accurate to within 7 units of the least significant three
;       bits of the NPX extended real format.  The PLM/86 definition is:
;
;       sine:   procedure (angle) real external;
;               declare angle real;
;               end sine;
;
;       cosine: procedure (angle) real external;
;               declare angle real;
;               end cosine;
;
;         Three stack registers are required.  The result of the function is
;       defined as follows for the following arguments:
;
;                       angle                                           result
;
;               valid or unnormal less than 2**62 in magnitude  correct value
;               zero                                            0 or 1
;               denormal                                        correct denormal
;               valid or unnormal greater than 2**62            indefinite
;               infinity                                        indefinite
;               NAN                                             NAN
;               empty                                           empty

;
;         This function is based on the NPX fptan instruction.  The fptan
;       instruction will only work with an angle of from 0 to PI/4.  With this
;       instruction, the sine or cosine of angles from 0 to PI/4 can be accurately
;       calculated.  The technique used by this routine can calculate a general
;       sine or cosine by using one of four possible operations:
;
;                       Let R = |angle mod PI/4|
;                           S = -1 or 1, according to the sign of the angle
;
;       1) sin(R)       2) cos(R)       3) sin(PI/4-R)  4) cos(PI/4-R)
;
;         The choice of the relation and the sign of the result follows the
;       decision table shown below based on the octant the angle falls in:
;
;               octant          sine            cosine
;
;                 0              S*1               2
;                 1              S*4               3
;                 2              S*2            -1*1
;                 3              S*3            -1*4
;                 4             -S*1            -1*2
;                 5             -S*4            -1*3
;                 6             -S*2               1
;                 7             -S*3               4
;

;
;         Angle to sine function is a zero or unnormal.
;
sine_zero_unnormal:

        fstp    st(1)                           ; Remove PI/4
        jnz     enter_sine_normalize            ; Jump if angle is unnormal
;
;         Angle is a zero.
;
        pop     bp                              ; Return the zero as the result
        ret
;
;         Angle is an unnormal.
;
enter_sine_normalize:

```

```
146             call    normalize_value
147             jmp     short enter_sine
148
149     cosine  proc                                    ; Entry point to cosine
150
151             fxam                                    ; Look at the value
152             push    bp                              ; Establish stack addressibility
153             sub     sp,size local_area              ; Allocate stack space for status
154             mov     bp,sp
155             fstsw   status                          ; Store status value
156             fld     pi_quarter                      ; Setup for angle reduce
157             mov     cl,1                            ; Signal cosine function
158             pop     ax                              ; Get status value
159             lahf                                    ; ZF = C3, PF = C2, CF = C0
160             jc      funny_parameter                 ; Jump if parameter is
161                                                     ; empty, NAN, or infinity
162     ;
163     ;         Angle is unnormal, normal, zero, denormal.
164     ;
165             fxch                                    ; st(0) = angle, st(1) = PI/4
166             jpe     enter_sine                      ; Jump if normal or denormal
167     ;
168     ;         Angle is an unnormal or zero.
169     ;
170             fstp    st(1)                           ; Remove PI/4
171             jnz     enter_sine_normalize
172     ;
173     ;         Angle is a zero.  cos(0) = 1.0
174     ;
175             fstp    st(0)                           ; Remove 0
176             pop     bp                              ; Restore stack
177             fld1                                    ; Return 1
178             ret
179     ;
180     ;         All work is done as a sine function.  By adding PI/2 to the angle
181     ;       a cosine is converted to a sine.  Of course the angle addition is not
182     ;       done to the argument but rather to the program logic control values.
183     ;
184     sine:                                           ; Entry point for sine function
185
186             fxam                                    ; Look at the parameter
187             push    bp                              ; Establish stack addressibility
188             sub     sp,size local_area              ; Allocate local space
189             mov     bp,sp
190             fstsw   status                          ; Look at fxam status
191             fld     pi_quarter                      ; Get PI/4 value
192             pop     ax                              ; Get fxam status
193             lahf                                    ; CF = C0, PF = C2, ZF = C3
194             jc      funny_parameter                 ; Jump if empty, NAN, or infinity
195     ;
196     ;           Angle is unnormal, normal, zero, or denormal.
197     ;
198             fxch                                    ; ST(1) = PI/4, st(0) angle
199             mov     cl,0                            ; Signal sine
200             jpo     sine_zero_unnormal              ; Jump if zero or unnormal
201     ;
202     ;         ST(0) is either a normal or denormal value.  Both will work.
203     ;       Use the fprem instruction to accurately reduce the range of the given
204     ;       angle to within 0 and PI/4 in magnitude.  If fprem cannot reduce the
205     ;       angle in one shot, the angle is too big to be meaningful, > 2**62
206     ;       radians.  Any roundoff error in the calculation of the angle given
207     ;       could completely change the result of this function.  It is safest to
208     ;       call this very rare case an error.
209     ;
210     enter_sine:
211
212             fprem                                   ; Reduce angle
213                                                     ; Note that fprem will force a
214                                                     ; denormal to a very small unnormal
215                                                     ; Fptan of a very small unnormal
216                                                     ; will be the same very small
217                                                     ; unnormal, which is correct.
218             mov     sp,bp                           ; Allocate stack space for status
219             fstsw   status                          ; Check if reduction was complete
```

```
220                                                     ; Quotient in C0,C3,C1
221             pop     bx                              ; Get fprem status
222             test    bh,high(mask cond2)             ; sin(2*N*PI+x) = sin(x)
223             jnz     angle_too_big
224     ;
225     ;         Set sign flags and test for which eighth of the revolution the
226     ;       angle fell into.
227     ;
228     ;       Assert: -PI/4 < st(0) < PI/4
229     ;
230             fabs                                    ; Force the argument positive
231                                                     ; condl bit in bx holds the sign
232             or      cl,cl                           ; Test for sine or cosine function
233             jz      sine_select                     ; Jump if sine function
234     ;
235     ;         This is a cosine function.  Ignore the original sign of the angle
236     ;       and add a quarter revolution to the octant id from the fprem instruction.
237     ;       cos(A) = sin(A+PI/2) and cos(|A|) = cos(A)
238     ;
239             and     ah,not high(mask condl)         ; Turn off sign of argument
240             or      bh,high(mask busy)              ; Prepare to add 010 to C0,C3,C1
241                                                     ; status value in ax
242                                                     ; Set busy bit so carry out from
243             add     bh,high(mask cond3)             ; C3 will go into the carry flag
244             mov     al,0                            ; Extract carry flag
245             rcl     al,1                            ; Put carry flag in low bit
246             xor     bh,al                           ; Add carry to C0 not changing
247                                                     ; C1 flag
248     ;
249     ;         See if the argument should be reversed, depending on the octant in
250     ;       which the argument fell during fprem.
251     ;
252     sine_select:
253
254             test    bh,high(mask condl)             ; Reverse angle if C1 = 1
255             jz      no_sine_reverse
256     ;
257     ;         Angle was in octants 1,3,5,7.
258     ;
259             fsub                                    ; Invert sense of rotation
260             jmp     short do_sine_fptan             ; 0 < arg <= PI/4
261     ;
262     ;         Angle was in octants 0,2,4,6.
263     ;       Test for a zero argument since fptan will not work if st(0) = 0
264     ;
265     no_sine_reverse:
266
267             ftst                                    ; Test for zero angle
268             mov     sp,bp                           ; Allocate stack space
269             fstsw   status                          ; cond3 = 1 if st(0) = 0
270             fstp    st(1)                           ; Remove PI/4
271             pop     cx                              ; Get ftst status
272             test    ch,high(mask cond3)             ; If C3=1, argument is zero
273             jnz     sine_argument_zero
274     ;
275     ;       Assert: 0 < st(0) <= PI/4
276     ;
277     do_sine_fptan:
278
279             fptan                                   ; TAN ST(0) = ST(1)/ST(0) = Y/X
280
281     after_sine_fptan:
282
283             pop     bp                              ; Restore stack
284             test    bh,high(mask cond3 + mask condl); Look at octant angle fell into
285             jpo     X_numerator                     ; Calculate cosine for octants
286                                                     ; 1,2,5,6
287     ;
288     ;         Calculate the sine of the argument.
289     ;       sin(A) = tan(A)/sqrt(1+tan(A)**2)       if tan(A) = Y/X then
290     ;       sin(A) = Y/sqrt(X*X + Y*Y)
291     ;
292             fld     st(1)                           ; Copy Y value
293             jmp     short finish_sine               ; Put Y value in numerator
```

```
294     ;
295     ;         The top of the stack is either NAN, infinity, or empty.
296     ;
297     funny_parameter:
298
299             fstp    st(0)                           ; Remove PI/4
300             jz      return_empty                    ; Return empty if no parm
301
302             jpo     return_NAN                      ; Jump if st(0) is NAN
303     ;
304     ;       st(0) is infinity.  Return an indefinite value.
305     ;
306             fprem                                   ; ST(1) can be anything
307
308     return_NAN:
309     return_empty:
310
311             pop     bp                              ; Restore stack
312             ret                                     ; Ok to leave fprem running
313     ;
314     ;         Simulate fptan with st(0) = 0
315     ;
316     sine_argument_zero:
317
318             fld1                                    ; Simulate tan(0)
319             jmp     after_sine_fptan                ; Return the zero value
320     ;
321     ;         The angle was too large.  Remove the modulus and dividend from the
322     ;       stack and return an indefinite result.
323     ;
324     angle_too_big:
325
326             fcompp                                  ; Pop two values from the stack
327             fld     indefinite                      ; Return indefinite
328             pop     bp                              ; Restore stack
329             fwait                                   ; Wait for load to finish
330             ret
331     ;
332     ;         Calculate the cosine of the argument.
333     ;       cos(A) = 1/sqrt(1+tan(A)**2)    if tan(A) = Y/X then
334     ;       cos(A) = X/sqrt(X*X + Y*Y)
335     ;
336     X_numerator:
337
338             fld     st(0)                           ; Copy X value
339             fxch    st(2)                           ; Put X in numerator
340
341     finish_sine:
342
343             fmul    st,st(0)                        ; Form X*X + Y*Y
344             fxch
345             fmul    st,st(0)
346             fadd                                    ; st(0) = X*X + Y*Y
347             fsqrt                                   ; st(0) = sqrt(X*X + Y*Y)
348
349     ;
350     ;         Form the sign of the result.  The two conditions are the C1 flag from
351     ;       FXAM in bh and the C0 flag from fprem in ah.
352     ;
353             and     bh,high(mask cond0)             ; Look at the fprem C0 flag
354             and     ah,high(mask cond1)             ; Look at the fxam C1 flag
355             or      bh,ah                           ; Even number of flags cancel
356             jpe     positive_sine                   ; Two negatives make a positive
357
358             fchs                                    ; Force result negative
359
360     positive_sine:
361
362             fdiv                                    ; Form final result
363             ret                                     ; Ok to leave fdiv running
364
365     cosine  endp
366
```

```
;
;         This function will calculate the tangent of an angle.
;       The angle, in radians is passed in ST(0), the tangent is returned
;       in ST(0).  The tangent is calculated to an accuracy of 4 units in the
;       least three significant bits of an extended real format number.  The
;       PLM/86 calling format is:
;
;     tangent:  procedure (angle) real external;
;               declare angle real;
;               end tangent;
;
;         Two stack registers are used.  The result of the tangent function is
;       defined for the following cases:
;
;                                  angle                              result
;
;               valid or unnormal < 2**62 in magnitude          correct value
;               0                                               0
;               denormal                                        correct denormal
;               valid or unnormal > 2**62 in magnitude          indefinite
;               NAN                                             NAN
;               infinity                                        indefinite
;               empty                                           empty
;
;         The tangent instruction uses the fptan instruction.  Four possible
;       relations are used:
;
;         Let R = |angle MOD PI/4|
;             S = -1 or 1 depending on the sign of the angle
;
;       1) tan(R)       2) tan(PI/4-R)  3) 1/tan(R)     4) 1/tan(PI/4-R)
;
;         The following table is used to decide which relation to use depending
;       on in which octant the angle fell.
;
;       octant          relation
;
;         0              S*1
;         1              S*4
;         2             -S*3
;         3             -S*2
;         4              S*1
;         5              S*4
;         6             -S*3
;         7             -S*2
;
tangent proc

        fxam                                    ; Look at the parameter
        push    bp                              ; Establish stack addressibility
        sub     sp,size local_area              ; Allocate local variable space
        mov     bp,sp
        fstsw   status                          ; Get fxam status
        fld     pi_quarter                      ; Get PI/4
        pop     ax
        lahf                                    ; CF = C0, PF = C2, ZF = C3
        jc      funny_parameter
;
;         Angle is unnormal, normal, zero, or denormal.
;
        fxch                                    ; st(0) = angle, st(1) = PI/4
        jpe     tan_zero_unnormal
;
;         Angle is either an normal or denormal.
;       Reduce the angle to the range -PI/4 < result < PI/4.
;       If fprem cannot perform this operation in one try, the magnitude of the
;       angle must be > 2**62.  Such an angle is so large that any rounding
;       errors could make a very large difference in the reduced angle.
;       It is safest to call this very rare case an error.
;
tan_normal:

        fprem                                   ; Quotient in C0,C3,C1
                                                ; Convert denormals into unnormals
```

```
441             mov     sp,bp                           ; Allocate stack spce
442             fstsw   status                          ; Quotient identifies octant
443                                                     ; original angle fell into
444             pop     bx                              ; tan(PI*N+x) = tan(x)
445             test    bh,high(mask cond2)             ; Test for complete reduction
446             jnz     angle_too_big                   ; Exit if angle was too big
447     ;
448     ;         See if the angle must be reversed.
449     ;
450     ;       Assert: -PI/4 < st(0) < PI/4
451     ;
452             fabs                                    ; 0 <= st(0) < PI/4
453                                                     ; C1 in bx has the sign flag
454             test    bh,high(mask cond1)             ; must be reversed
455             jz      no_tan_reverse
456     ;
457     ;         Angle fell in octants 1,3,5,7.  Reverse it, subtract it from PI/4.
458     ;
459             fsub                                    ; Reverse angle
460             jmp     short do_tangent
461     ;
462     ;         Angle is either zero or an unnormal.
463     ;
464     tan_zero_unnormal:
465
466             fstp    st(1)                           ; Remove PI/4
467             jz      tan_angle_zero
468     ;
469     ;         Angle is an unnormal.
470     ;
471             call    normalize_value
472             jmp     tan_normal
473
474     tan_angle_zero:
475
476             pop     bp                              ; Restore stack
477             ret
478     ;
479     ;         Angle fell in octants 0,2,4,6.  Test for st(0) = 0, fptan won't work.
480     ;
481     no_tan_reverse:
482
483             ftst                                    ; Test for zero angle
484             mov     sp,bp                           ; Allocate stack space
485             fstsw   status                          ; C3 = 1 if st(0) = 0
486             fstp    st(1)                           ; Remove PI/4
487             pop     cx                              ; Get ftst status
488             test    ch,high(mask cond3)
489             jnz     tan_zero
490
491     do_tangent:
492
493             fptan                                   ; tan ST(0) = ST(1)/ST(0)
494
495     after_tangent:
496     ;
497     ;         Decide on the order of the operands and their sign for the divide
498     ;       operation while the fptan instruction is working.
499     ;
500             pop     bp                              ; Restore stack
501             mov     al,bh                           ; Get a copy of fprem C3 flag
502             and     ax,mask cond1 + high(mask cond3); Examine fprem C3 flag and
503                                                     ; fxtract C1 flag
504             test    bh,high(mask cond1 + mask cond3); Use reverse divide if in
505                                                     ; octants 1,2,5,6
506             jpo     reverse_divide                  ; Note! parity works on low
507                                                     ;  8 bits only!
508     ;
509     ;         Angle was in octants 0,3,4,7.
510     ;       Test for the sign of the result.  Two negatives cancel.
511     ;
512             or      al,ah
513             jpe     positive_divide
```

```
514
515             fchs                                    ; Force result negative
516
517     positive_divide:
518
519             fdiv                                    ; Form result
520             ret                                     ; Ok to leave fdiv running
521
522     tan_zero:
523
524             fldl                                    ; Force 1/0 = tan(PI/2)
525             jmp     after_tangent
526     ;
527     ;         Angle was in octants 1,2,5,6.
528     ;       Set the correct sign of the result.
529     ;
530     reverse_divide:
531
532             or      al,ah
533             jpe     positive_r_divide
534
535             fchs                                    ; Force result negative
536
537     positive_r_divide:
538
539             fdivr                                   ; Form reciprocal of result
540             ret                                     ; Ok to leave fdiv running
541
542     tangent endp
543     ;
544     ;         This function will normalize the value in st(0).
545     ;       Then PI/4 is placed into st(1).
546     ;
547     normalize_value:
548
549             fabs                                    ; Force value positive
550             fxtract                                 ; 0 <= st(0) < 1
551             fldl                                    ; Get normalize bit
552             fadd    st(1),st                        ; Normalize fraction
553             fsub                                    ; Restore original value
554             fscale                                  ; Form original normalized value
555             fstp    st(1)                           ; Remove scale factor
556             fld     pi_quarter                      ; Get PI/4
557             fxch
558             ret
559
560     code    ends
561             end

ASSEMBLY COMPLETE, NO ERRORS FOUND
```